NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

INFORMATION SECURITY CONSIDERATIONS FOR APPLICATIONS USING APACHE ACCUMULO

by

Brandon H. Pontius

September 2014

Thesis Advisor: Mark Gondree
Second Reader: Garrett McGrath

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave Blank)*	**2. REPORT DATE** 09-26-2014	**3. REPORT TYPE AND DATES COVERED** Master's Thesis 08-27-2012 to 09-26-2014

4. TITLE AND SUBTITLE

INFORMATION SECURITY CONSIDERATIONS FOR APPLICATIONS USING APACHE ACCUMULO

5. FUNDING NUMBERS

6. AUTHOR(S)

Brandon H. Pontius

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

NoSQL databases are gaining popularity due to their ability to store and process large heterogeneous data sets more efficiently than relational databases. Apache Accumulo is a NoSQL database that introduced a unique information security feature—cell-level access control. We study Accumulo to examine its cell-level access control policy enforcement mechanism. We survey existing Accumulo applications, focusing on Koverse as a case study to model the interaction between Accumulo and a client application. We conclude with a discussion of potential security concerns for Accumulo applications. We argue that Accumulo's cell-level access control can assist developers in creating a stronger information security policy, but Accumulo cannot provide security—particularly enforcement of information flow policies—on its own. Furthermore, popular patterns for interaction between Accumulo and its clients require diligence on the part of developers, which may otherwise lead to unexpected behavior that undermines system policy. We highlight some undesirable but reasonable confusions stemming from the semantic gap between cell-level and table-level policies, and between policies for end-users and Accumulo clients.

14. SUBJECT TERMS

big data, NoSQL, databases, information security, Accumulo, cell-level security

15. NUMBER OF PAGES 85

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

THIS PAGE INTENTIONALLY LEFT BLANK

INFORMATION SECURITY CONSIDERATIONS FOR APPLICATIONS USING APACHE ACCUMULO

Brandon H. Pontius
Lieutenant, United States Navy
B.S., Louisiana State University, 2005
M.B.A., Louisiana State University, 2012

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2014

Author:	Brandon H. Pontius
Approved by:	Mark Gondree, Ph.D. Thesis Advisor
	Garrett McGrath Second Reader
	Peter Denning, Ph.D. Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

NoSQL databases are gaining popularity due to their ability to store and process large heterogeneous data sets more efficiently than relational databases. Apache Accumulo is a NoSQL database that introduced a unique information security feature—cell-level access control. We study Accumulo to examine its cell-level access control policy enforcement mechanism. We survey existing Accumulo applications, focusing on Koverse as a case study to model the interaction between Accumulo and a client application. We conclude with a discussion of potential security concerns for Accumulo applications. We argue that Accumulo's cell-level access control can assist developers in creating a stronger information security policy, but Accumulo cannot provide security—particularly enforcement of information flow policies—on its own. Furthermore, popular patterns for interaction between Accumulo and its clients require diligence on the part of developers, which may otherwise lead to unexpected behavior that undermines system policy. We highlight some undesirable but reasonable confusions stemming from the semantic gap between cell-level and table-level policies, and between policies for end-users and Accumulo clients.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

API	Application Programming Interface
ACID	Atomicity Consistency Isolation Durability
BASE	Basically Available Soft-state Eventually Consistent
DAC	Discretionary Access Control
DARPA	Defense Advanced Research Projects Agency
DOD	Department of Defense
HDFS	Hadoop Distributed File System
INSCOM	United States Army Intelligence and Security Command
JPA	Java Persistence API
JSON	JavaScript Object Notation
MAC	Mandatory Access Control
MUSE	Mining and Understanding Software Enclaves
NoSQL	Not Only SQL
NSA	National Security Agency
NTC	Naval Tactical Cloud
ONR	Office of Naval Research
SQL	Structured Query Language
TCSEC	Trusted Computer System Evaluation Criteria
UCD	Unified Cloud Data

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

I want to thank my thesis advisor, Dr. Mark Gondree, for his guidance. He always provided just the direction I needed to keep me on track and focused during research and writing.

I also want to thank Garrett McGrath for taking the time to be my second reader and making sure my writing made sense.

Finally, and most importantly, I want to thank my wife. Anne, you have always supported me in anything I do, and I would not have been able to complete this program without you. You are an incredible mother to our children, and throughout our time in Monterey, you have gone above and beyond to make sure everything at home is taken care of so I could have time to complete all of my academic requirements. I love you, and I am grateful every day for the life we have been blessed with.

To my children: You will probably never read this. But if you do, know that I love you, and you bring joy to my life every day.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1: Introduction

For decades, relational databases have been the preferred method for readily retrievable data storage. As data sets have become larger and less structured, inefficiencies have emerged with relational databases [1]. The desire to solve these problems led to the development of Not Only SQL (NoSQL) databases. Their popularity has grown rapidly during the last ten years, and NoSQL databases are now used by several large companies, such as Google, Facebook, Twitter, LinkedIn, Amazon and others, to manage large data sets.

Accumulo is a NoSQL database developed by the government primarily to store and process large amounts of intelligence data [2]. The Accumulo project was an early developer of cell-level access control for NoSQL databases. Recently, other NoSQL projects such as HBase have followed suit. Cell-level access control is designed to allow secure access to data sets of mixed sensitivity levels. This work attempts to describe the technical aspects of Accumulo's cell-level access control policy enforcement and comment more generally on Accumulo's role in maintaining data security in production applications.

1.1 Big Data in the Military

The amount of data human beings generate and consume is increasing exponentially in both the commercial sector and in the Department of Defense (DOD). In 2012, it was estimated that seven million computing devices were being used in the military to process a 1,600 percent increase in data since September 2011 [3]. Currently, there are between two and five terabytes of data stored for each member of the armed services [4]. Generation of large amounts of data does not necessarily translate to good intelligence as analysts can become overwhelmed by the volume of data. An analyst attempting to glean intelligence from modern data streams has been compared to a person trying to quench his thirst with a fire hose. [3]. Attempting to process so much data creates an "operational thrashing" problem in which analysts spend more time organizing and preprocessing data than creating actionable intelligence [5]. To understand the amount of data a typical analyst may have to sift through, consider sitting down at a computer and looking through hundreds of thousands of spreadsheets each with hundreds of columns and tens of thousands of rows [6]. In a 2012

Forbes article, Lt. Gen. Michael Oates, head of the Joint Improvised Explosive Device Organization, commented, “There is no shortage of data. There is a dearth of analysis” [3].

Generation of actionable intelligence from large data sets requires efficient analysis. Manual analysis of large data sets to develop these insights is unsustainably resource intensive. In January 2014, the deputy director of the Defense Intelligence Agency noted, “We’re looking for needles within haystacks while trying to define what the needle is, in an era of declining resources and increasing threats” [7]. Big data platforms have the storage and analytical capabilities necessary to handle large data sets. These solutions can relieve the processing burden on human analysts and allow them to spend more time generating real intelligence [5]. Big data analytics make information more usable, improve decision making, and lead to more focused missions and services. For instance, geographically separated teams can access a real-time common operating picture, diagnostic data mining can support proactive maintenance programs that prevent battlefield failures, and data can be transformed into a common structure that allows custom queries by a distributed force composed of many communities [4], [6].

Despite the constrained budgetary environment, the DOD continues to invest in big data. The DOD spends $250 million a year on big data initiatives, according to *MilitaryTimes*, and the FY2015 budget establishes big data investment among its science and technology priorities [7]. Several DOD agencies are funding big data programs. For example, the Defense Advanced Research Projects Agency (DARPA) MUSE program seeks to improve the software engineering process by mining a large corpus of software to find useful properties, behaviors, and vulnerabilities and leverage that information to increase software reliability [8]. The XDATA program, also backed by DARPA, is developing new computational methods and tools for processing big data sets [9]. The Office of Naval Research (ONR) Naval Tactical Cloud (NTC) project seeks to improve intelligence distribution across disparate forces using cloud technologies [10].

As the DOD develops technologies to analyze and distribute information more efficiently, data security becomes more of a concern. Data flowing through mobile devices and across land, sea, and air battle spaces creates more opportunities for adversaries to intercept or manipulate data [4]. Applications must be developed with these security concerns in mind.

1.2 Contributions

This thesis seeks to determine the role of Accumulo's cell-level security in applications requiring information security. Because Accumulo documentation does not provide a detailed description of its operation, we use static analysis of Accumulo source code to describe Accumulo's architecture and detail its cell-level access control policy enforcement. We discuss the interfaces between Accumulo and client applications. Finally, we describe potential security concerns for Accumulo based applications and argue that, while Accumulo provides some assistance to developers in maintaining data security, a significant portion of the overall security policy must be enforced at the client application level. We believe our technical survey may assist future study in identifying and mitigating potential information security vulnerabilities in Accumulo or Accumulo based applications. Our comments on potential concerns for configuration of Accumulo client and user interaction motivate the need for a more thorough "best practice" guide.

1.3 Thesis Organization

In Chapter 2, we provide background on NoSQL and Accumulo. Chapter 3 describes Accumulo's data model and software and hardware architecture. In Chapter 4, we discuss the use of *Authorizations* and *ColumnVisibilities* to enforce cell-level access control in Accumulo. This discussion includes a walk-through of those critical portions of Accumulo code used for policy enforcement. Chapter 5 provides a general overview of Accumulo client applications, and Chapter 6 provides a detailed discussion of Koverse as a case study. Chapter 7 is a discussion of potential security concerns for applications that integrate with Accumulo. Finally, in Chapter 8 we present conclusions and topics for further study.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Background

In this chapter, we provide an overview of the NoSQL ecosystem and NoSQL security concerns as well as a description of Accumulo's role in the NTC.

2.1 NoSQL Ecosystem

NoSQL databases are gaining popularity as developers seek to address problems with traditional relational databases. A 2012 Couchbase survey asked database system developers what they considered to be the most critical problems with relational databases that influenced their decision to use NoSQL solutions. Of the survey respondents, 49 percent identified rigid schemas as a significant problem, 39 percent said lack of scalability, and 29 percent said high latency [11]. NoSQL databases offer several benefits [12] over relational databases, including:

Reduced complexity. The rich feature set and strict ACID properties of relational databases may not be necessary for some data sets.

Higher throughput. Cassandra writes 2,500 times faster into a 50GB database than MySQL [13]. BigTable can process 20 petabytes per day [14].

High degree of scalability on commodity hardware. NoSQL databases do not rely on highly available hardware and are designed to handle failure efficiently. Data can be partitioned across hardware more efficiently than relational database sharding. Hardware nodes can be added and removed relatively easily.

More flexible data model. NoSQL databases are not restricted to the relational data model which can be inefficient for unstructured data sets.

While NoSQL databases address some problems with the relational model, they also present their own set of problems. Most notable is the weaker guarantees offered by NoSQL databases compared to ACID systems. Brewer's CAP theorem says that database systems must balance consistency, availability, and partition tolerance and that strong forms of all three properties cannot be achieved simultaneously [15], [16]. NoSQL databases generally sacrifice consistency for increased availability and partition tolerance. In contrast to ACID properties provided by relational databases, many NoSQL systems claim to provide BASE properties—basically available, soft-state, eventually consistent [17]. Another weakness of NoSQL databases is the lack of a common interface like Structured Query Language (SQL). SQL simplifies and standardizes database manipulation in relational databases. NoSQL databases each have a unique programming interface that uses a lower level procedural language (e.g., Java) and requires more complex programming than SQL to perform the same task [18].

Although NoSQL solutions are becoming a larger presence in the database community, relational databases continue to be far more prevalent. Table 2.1 shows the ten most used databases along with several other NoSQL databases for comparison, as reported by DB-Engines. According to DB-Engines, the scores are standardized such that a database with twice the score is twice as popular. MongoDB and Cassandra are the only two NoSQL databases in the top ten and are much less popular than the top relational databases, but NoSQL database use is increasing [19], as shown in Figure 2.1. The 2012 Couchbase study claimed that 70 percent of large companies planned to fund NoSQL projects in 2012. Forty percent of companies surveyed said that NoSQL technologies were important or critical to daily operations, and an additional 37 percent said NoSQL was becoming important [11].

There are many types and implementations of NoSQL databases, but most share some common features. The most obvious is that they do not conform to the relational data model and are not heavily dependent on tables of data, or any other particular schema. They also use a lower level procedural query interface rather than SQL. Finally, NoSQL databases scale well horizontally by distributing data across a “nothing shared” network of commodity hardware [17], [18]. NoSQL databases are designed to perform in a variety of use cases including large volume data storage, large scale data processing, embedded (machine-to-machine) information retrieval, and exploratory analytics [12].

Rank	Database	Score
1	Oracle	1470.86
2	MySQL	1281.22
3	Microsoft SQL Server	1242.50
4	PostgreSQL	249.85
5	MongoDB	237.36
6	DB2	206.42
7	Microsoft Access	139.62
8	SQLite	88.87
9	Sybase ASE	86.17
10	Cassandra	81.90
11	Redis	70.80
15	HBase	41.92
18	Memcached	30.99
21	CouchDB	24.13
30	Riak	11.67
54	Accumulo	2.62

Table 2.1: Popularity of NoSQL databases, as reported by DB-Engines August 2014 rankings, after [19].

NoSQL databases are grouped in three categories. Key-value stores are the simplest of the NoSQL implementations. They store data in maps, dictionaries, or hash tables [17] and use basic put and get operations to write and read entries by key. The value is not searchable. Key-value stores feature high scalability and efficient retrieval but lack complex querying capability [12]. Examples of key-value stores are Dynamo, Voldemort, Redis, Riak, and Memcached [12], [18]. Document stores add a level of complexity to simple key-value stores. These NoSQL databases store documents [12], typically in a standard data exchange format such as XML, JSON, or BSON [17]. Key-value pairs are encapsulated in these schemaless documents. Both keys and values are searchable [17]. MongoDB and CouchDB are the most common examples of document stores [18]. Column-oriented stores are modeled after Google's BigTable design. They store and process data by column and the keys have multiple attributes. They often integrate with a distributed file system such as Google File System or Hadoop Distributed File System and a data analytic framework such as MapReduce [17]. Examples of column-oriented stores are BigTable, HBase, Hypertable, Cassandra, and Accumulo [18].

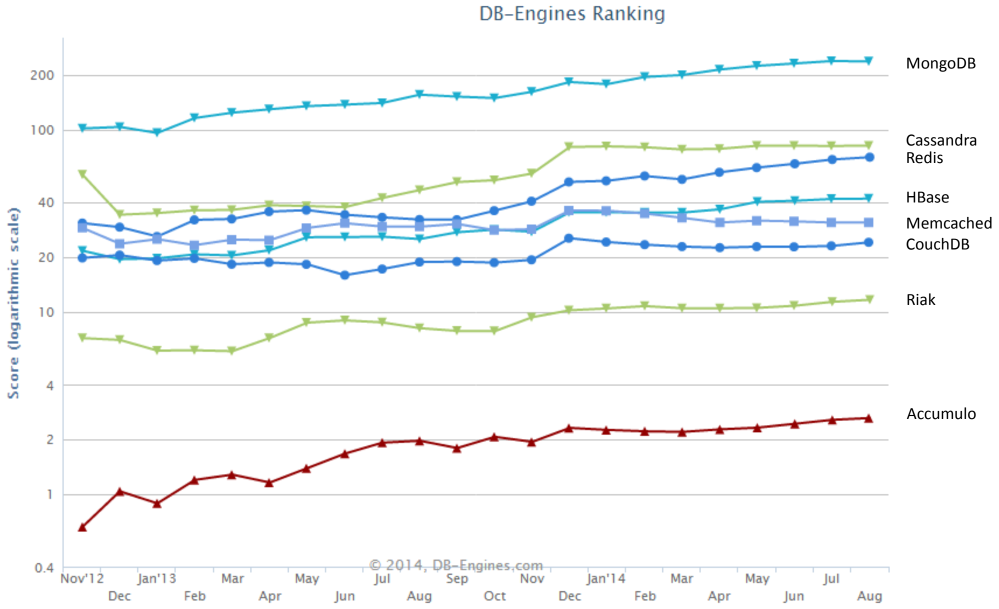


Figure 2.1: Trends in NoSQL database popularity, from [19].

Accumulo's design is based on Google's BigTable [20]. The data model and technology dependencies are two major aspects of BigTable that carry into Accumulo. BigTable introduced a multi-attribute key that identifies a row, column family, column qualifier, and timestamp with each data entry. Entries are stored in *Tables* which are distributed across commodity hardware by dividing them into subsets call *Tablets*. A *Tablet Server* process runs on each BigTable node that manages a set of *Tablets*. BigTable uses a distributed file system, Google File System, for persistent storage, integrates with the MapReduce analytic framework, and uses a distributed service to manage concurrency and consistency of distributed nodes. All of these properties are also present in Accumulo and are discussed in more detail in later chapters.

2.2 NoSQL Security

As the scale of information sharing grows, so does the problem of maintaining the security of that information. The growing numbers of information users combined with more direct access to data requires closer attention to security policies and their enforcement. The wide use of web interfaces to applications with database backends illustrates this problem. Users are given more access through these interfaces, which are frequent victims of cyber attacks. Databases have an important role in maintaining the confidentiality, integrity, and availability of data. A compromised database can lead to improper access to data, improper modification of data, or loss of access to data. These problems affect not only the individual that owns the compromised data, but entire organizations and communities [21].

Okman *et al.* investigated NoSQL security in more detail using Cassandra and MongoDB [22]. They identified the following potential security weaknesses:

- No encryption mechanism for data
- Unencrypted communication with clients
- Usernames and passwords sent as clear text
- Option available to encrypt inter-node communication but not the default setting
- No protection during bulk data ingest
- Query languages potentially susceptible to injection attacks
- Denial of service by thread consumption
- Weak native authentication and authorization implementations
- No redundancy in password and permission files
- Permission files not verified during each request

The Cloud Security Alliance defines the most critical information security threats they perceive for big data, grouping these into the following categories [23]:

1. Secure computations in distributed programming frameworks
2. Security best practices for non-relational databases
3. Privacy preserving data mining and analytics
4. Cryptographically enforced data centric security
5. Granular access control
6. Secure data storage and transaction logs
7. Granular audits
8. Data provenance
9. End point validation and filtering

10. Real-time security monitoring

Of these concerns, the most relevant to our discussion of Accumulo are:

Security best practices for non-relational data stores

NoSQL databases have been designed with performance in mind and with few built in security features. Much of the NoSQL community relies on middleware to enforce security policy. Each NoSQL solution has a unique interface, so developers face the challenge of verifying the correctness of middleware security protocols and ensuring proper integration with a specific NoSQL database.

Granular access control

Big data, in an operational or intelligence context, originates from a variety of sources and sensitivity levels. Coarse access control policies may unnecessarily restrict information that could be used to generate insightful analytics. Finer access control, such as Accumulo's cell-level control, can maximize data sharing while maintaining secrecy.

2.3 Naval Tactical Cloud

The Unified Cloud Data (UCD) ecosystem was developed by United States Army Intelligence and Security Command (INSCOM) to improve data sharing and analytic capabilities. The NTC project seeks to adapt the UCD model for use by the Navy [10]. NTC addresses military information dissemination challenges including distribution of data over a tactical force, prioritizing data movement in constrained network conditions, representation of data for efficient movement across tactical networks, prioritizing data retention and indexes in constrained storage conditions, and designing analytics that work across a distributed force. NTC plans to meet these challenges by combing semantic web and big data technologies to merge data sets from different communities leading to more insightful, actionable intelligence.

Accumulo is an integral part of the NTC architecture. NTC data is represented in a graph structure that defines relationships between data items. This structure makes it easy to add new data or merge disjoint data sets. This data model requires the addition of metadata to identify nodes of the graph, their properties, and the relationships between them. NTC uses

Accumulo to provide distributed storage of raw data items and all metadata necessary to integrate data into the graph.

Graph edges are three tuples that identify a subject, object, and a relationship between them. Subjects and objects can be any entity within the context of the data that the graph describes. These are referred to collectively as *Terms* and are stored together in an Accumulo *Term Table*. Relationships are stored in a separate *Predicate Table*. The *Statement Table* stores the graph edges via the subject-object-predicate tuples. An *Artifact Table* preserves the raw input data items prior to graph processing [10, pp. 80-88].

Accumulo was chosen for this task, at least in part, because of its cell-level access control capability. Fine-grained access control could enhance data availability and thereby enhance analytical processing and information dissemination while maintaining information security. Unfortunately, the NTC project is still under development and a detailed description of how Accumulo's cell-level access control would be used in NTC is not available.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Accumulo Overview

Accumulo is a distributed data storage application developed by the National Security Agency (NSA), following and extending Google's BigTable design [20]. Under pressure from the United States Senate Armed Services Committee, the NSA submitted Accumulo as an open source project that is now run by Apache [24]. Accumulo is a NoSQL database, a term used to describe a large family of data storage solutions that do not adhere to a traditional relational database model. Like other NoSQL databases, Accumulo provides a simple and flexible data model with restricted query semantics. This simplicity is credited as enabling scalability, handling large data sets while maintaining efficient data retrieval performance. Benchmarking studies have shown Accumulo to be capable of processing hundreds of terabytes of data at rates of over 100 million data entries per second [25]–[27]. In contrast to similar column-oriented NoSQL data stores, Accumulo adds cell-level access control to its data retrieval model. This chapter describes Accumulo's data model and system architecture.

3.1 Data Model

While Accumulo is a column-oriented store, like Google BigTable and Apache HBase, it can be viewed as a simple key-value store. The key is composed of five different elements: row, column family, column qualifier, column visibility, and a timestamp (see Figure 3.1).

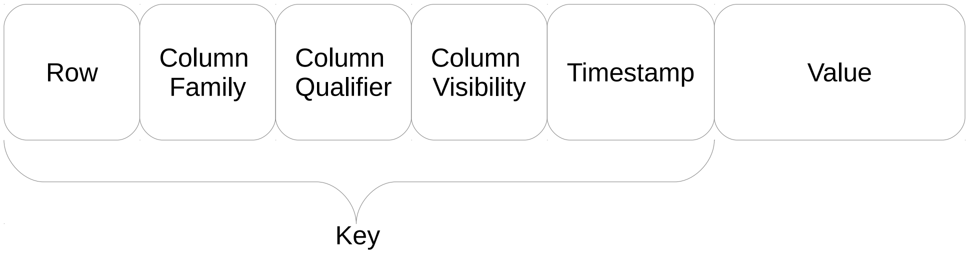


Figure 3.1: Accumulo key-value relationship

The row, column family, and column qualifier elements are used to uniquely identify a set of timestamped values in Accumulo. All information that will be used to locate a specific value must be encoded in these three elements of the key. The column visibility element is used to enforce Accumulo's cell-level access control. Clients present a set of authorizations to Accumulo, which it uses to filter data it returns to the client based on the policy in each column visibility element. We discuss the interaction between authorizations and column visibilities in more detail in Chapter 4. The timestamp element is used to implement cell-level versioning. Any entries with identical row, column family, and column qualifier elements are assumed to be different versions of the same value field. By default, Accumulo returns only the most recent version of an entry. The value element is the raw data stored in Accumulo. All elements of the key-value pair are stored as byte arrays with the exception of the timestamp, which is stored as an integer. This generic typing of key elements allows the Accumulo client flexibility in determining what data types will be used as each part of the key-value entry.

Accumulo automatically sorts data lexicographically by key upon ingest, so data with similar keys are stored together. This strategy allows efficient range queries to take advantage of data locality: related data, which is more likely to be accessed near the same time, is stored near each other, decreasing overall access time.

Accumulo groups sorted key-value pairs into tables. Tables are used to organize and distribute Accumulo entries across data storage nodes. Tables can be split along row boundaries into smaller subsets called tablets. Tablets are the basic data structures that are maintained by individual nodes in Accumulo's distributed architecture.

The combination of table, row, column family, and column qualifier can be used to apply a logical hierarchy to Accumulo data [28]. The key hierarchy is flexible and can be used to organize data in many ways, ranging from a traditional relational table framework to completely unstructured data. Figure 3.2 shows how the Accumulo key hierarchy might be used by an organization to store employee information. Each employee is represented as a row in the Employees Table. There is no requirement for each row in a table to have the same number or types of columns, so each employee could have different types of information stored. In this example, Bob does not have an office, so no location information is stored. In a traditional relational database, the Employees Table would have empty cells in Bob's location entries, resulting in inefficient use of space. The key-value data model allows flexible data organization as well as efficient data distribution across the individual nodes in Accumulo's architecture.

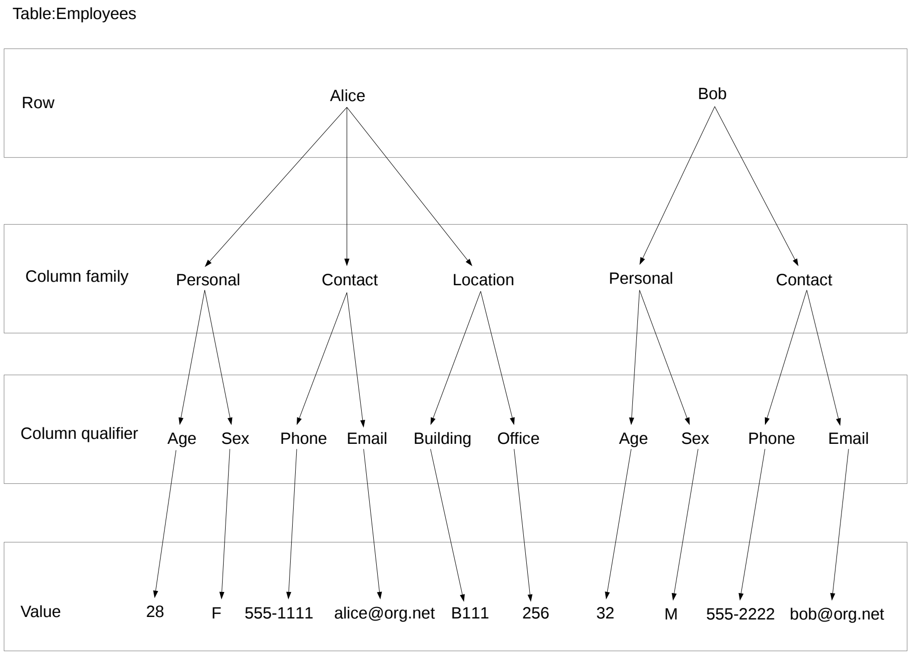


Figure 3.2: Accumulo key hierarchy

3.2 System Architecture

Accumulo relies on Hadoop Distributed File System (HDFS) and Zookeeper to provide data storage across distributed commodity hardware. HDFS provides Accumulo with distributed data persistence. Zookeeper manages coordination of concurrent distributed processes. Individual components of an Accumulo instance can run on separate machines in different geographic locations.

3.2.1 Accumulo Components

The main components of an Accumulo instance are a master server, a monitor, one or more tablet servers, a garbage collector, and one or more clients.

Master. The master is responsible for managing tablet servers. It ensures that each tablet is assigned to exactly one tablet server and that load is balanced across tablet servers.

It manages recovery in the event of a tablet server failure to ensure reliable persistence of tablets. It also handles table management requests (creation, modification, deletion) from clients.

Monitor. The monitor provides a web interface to monitor Accumulo performance. It is controlled by the master.

Tablet server. The tablet server is the main data management component of Accumulo. Each tablet server handles a subset of all tablets in the Accumulo instance. The main function of a tablet server is to handle read and write requests from clients. In response to a write request, the tablet server saves new data in memory in the *memtable* data structure, sorts key-value pairs in memory, and periodically writes sorted key-value pairs to HDFS for permanent storage. The tablet server also make entries about write events in a write-ahead log, to provide an efficient mechanism for tablet server failure recovery. In response to a read request, the tablet server provides to the client a sorted set of the requested key-value pairs, by merging data stored in HDFS and memory.

Garbage collector. The garbage collector ensures efficient use of HDFS storage space by identifying and deleting files that are no longer used by any process.

Client. Accumulo provides a client Application Programming Interface (API) that contains interfaces for connecting to an Accumulo instance and executing read and write requests.

3.2.2 HDFS Components

The main components of HDFS are a name node, a secondary name node, a job tracker, one or more data nodes and one or more task trackers.

Name node. The name node is the master process in HDFS. It controls the HDFS namespace and client access to HDFS files. It keeps track of where in HDFS each individual file is stored.

Secondary name node. The secondary name node tracks HDFS state information that is used by the name node at startup. It is not a backup for the name node.

Data node. The data nodes store files in HDFS.

Job tracker. The job tracker manages MapReduce jobs. It divides each job into tasks and assigns them to task trackers.

Task tracker. Task trackers perform work necessary to execute MapReduce jobs. They perform tasks assigned by the job tracker.

Each of the Accumulo and HDFS components are implemented by separate processes. Production implementations of Accumulo may co-locate these processes, if appropriate, depending on hardware, performance and availability. Although it is possible to run all Accumulo processes on one machine, an effective implementation will distribute workload across multiple machines [29].

3.2.3 Hardware Architecture

Accumulo uses a distributed network of hardware to provide scalable data storage. Figure 3.3 illustrates the interaction between Accumulo components in a possible distributed architecture. Each gray box indicates a separate physical machine, blue circles are processes, and green rectangles highlight notable data structures. Arrows indicate communication between physical components.

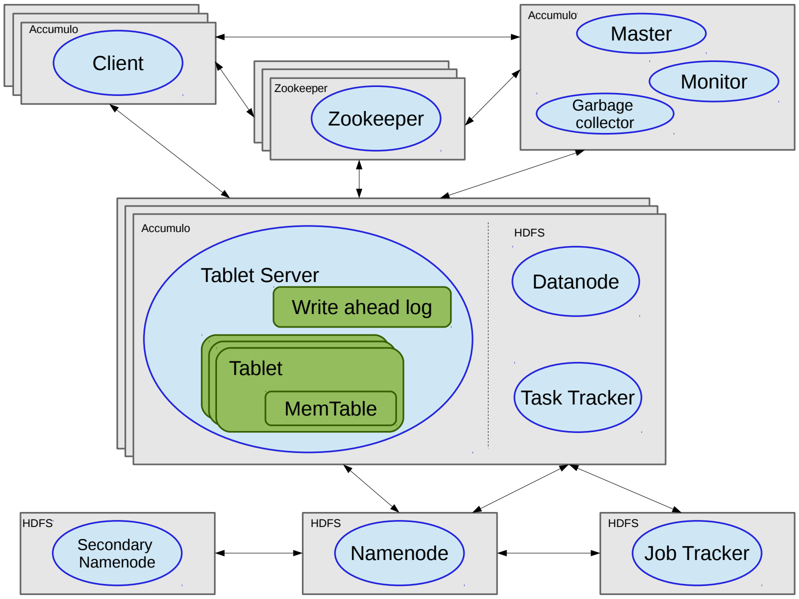


Figure 3.3: Accumulo architecture

Client machines communicate with Zookeeper and tablet servers to make read and write requests. Clients may communicate with the master to perform administrative tasks and table operations (e.g., table creation). Zookeeper maintains consistent configuration and status information for all tablet servers. The master communicates with the individual tablet servers to distribute tablet load and respond to tablet server failure, and communicates with Zookeeper to promulgate tablet server status. The namenode communicates with the tablet server to provide the location of data in HDFS. It manages individual datanodes to ensure proper data distribution throughout HDFS. The job tracker communicates with the individual task trackers to execute MapReduce jobs. The secondary namenode maintains state information for the namenode, to be used if the namenode is restarted.

CHAPTER 4:
Accumulo Cell-Level Policy Enforcement

Accumulo was the first to implement cell-level access control in the domain of NoSQL databases [30]. Databases generally grant user access permission at the table level [31], and in some cases, additional algorithms or data structures can be used to implement row or column level access control [32]. Accumulo's cell-level security is native functionality that gives system administrators tighter control of user access to data. With coarser data access control, an administrator may have to make a choice between data security and availability. If an entire table, column, or row is restricted, there may be information within that dataset that should be accessible but is restricted to keep the other data in the dataset secure. Accumulo cell-level access control provides flexibility that prohibits access to data in accordance with policy, while maximizing access to other data [33].

Accumulo's fine-grained access control is implemented by a column visibility label that is attached to each key-value pair. Clients that query the Accumulo database must provide a set of authorizations that are compared against column visibilities to determine if the client has access to each key-value pair. Accumulo only returns those entries that are accessible by the client. In this chapter we examine the process that Accumulo uses to enforce cell-level data access control.

4.1 Column Visibility

An Accumulo column visibility is a security label that is applied to each key-value pair. Although the column visibility is described in Accumulo documentation as an element of the key, it is not used to identify or locate data. Rather, it is an additional piece of metadata that is used to filter key-value pairs that are returned to the client. The visibility label is implemented as a Java *ColumnVisibility* object that becomes part of the key in each entry upon insertion. Within each *ColumnVisibility* object is a boolean expression that describes the authorizations needed to access the respective entry. A *ColumnVisibility* object stores the visibility expression in two ways. The first is a character string representing the raw boolean expression. The second is the root node of a binary tree describing the visibility expression. The *ColumnVisibility* object parses the visibility expression and generates a tree during initial construction of the object. Client code that queries the Accumulo database must present authorizations that satisfy the boolean expression in order to retrieve a particular entry.

4.1.1 *ColumnVisibility* Expression Syntax

The visibility expression is a boolean expression that describes a set of authorizations that must be provided to gain access to the data. The expression relates a set of tokens through logical conjunction and disjunction. Syntactically, tokens are represented by character strings, conjunction by the “&” character, and disjunction by the “|” character. Conjunctive phrases must be grouped separately from disjunctive phrases using parentheses to explicitly indicate precedence of operations. Beyond this minimum requirement, additional parentheses may be used as desired to group individual tokens or groups of tokens. Token strings in the visibility expression do not need to be quoted unless non-standard characters are required. Standard characters include alphanumerics, underscore, hyphen, colon, period, and frontslash. If the token is quoted, any characters can be used with the exception of backslash and double quotes. These characters must be prefaced by a backslash when used in quoted strings.

Figure 4.1 is a context-free grammar representation of the *ColumnVisiblity* expression syntax. Non-terminal symbols are enclosed in angled brackets. Terminal symbols are enclosed in single quotations. Braces indicate a set of ASCII character terminal symbols. Within braces, the carat represents a logical negation indicating that the subsequent characters are not part of the set. The dash indicates a range of ASCII characters. Table 4.1 provides examples of valid and invalid visibility strings.

<table>
<tr><th>Valid</th><th>Invalid</th></tr>
<tr><td>(one&two)|(three&four)</td><td>one&two|three</td></tr>
<tr><td>((A)&(B))|C|(D)</td><td>A!&B#</td></tr>
<tr><td>./a/:2-_-&:b:__-4/:</td><td>"1234&"&5678"</td></tr>
<tr><td>"#!\""|"*@\\%"</td><td>"abc\123"</td></tr>
</table>

Table 4.1: Examples of valid and invalid *ColumnVisibilities*

```
    <VISIBILITY> ::= '(' <VISIBILITY> ')'
                   | <TERM>
                   | <ANDS> '&' <ANDS>
                   | <ORS>  '|' <ORS>


          <ANDS> ::= '(' <ANDS> ')'
                   | <TERM>
                   | '(' <ORS> ')'
                   | <ANDS> '&' <ANDS>


           <ORS> ::= '(' <ORS> ')'
                   | <TERM>
                   | '(' <ANDS> ')'
                   | <ORS> '|' <ORS>


          <TERM> ::= '(' <TERM> ')'
                   | '"' <QUOTECHARS> '"'
                   | <NOQUOTECHARS>


    <QUOTECHARS> ::= <QCHAR>
                   | <QCHAR> <QUOTECHARS>


         <QCHAR> ::= [^\"]
                   | '\\'
                   | '\"'


  <NOQUOTECHARS> ::= NQCHAR
                   | NQCHAR NOQUOTECHARS


        <NQCHAR> ::= [a-zA-Z0-9_-:./]
```

Figure 4.1: *ColumnVisibility* expression syntax as a context free grammar

4.1.2 Parsing a *ColumnVisibility*

A *ColumnVisibility* object contains a parsing algorithm that is used to generate a binary tree from the boolean visibility expression. The tree is used to facilitate authorization checking during a query. The parser scans the expression left to right looking for “&” and “|” characters which become the root nodes of subtrees within the parse tree. The leaf nodes of the parse tree are the terms of the visibility expression.

Each node is a *Node* object containing three pieces of information: range, type, and children. The *Node* range is defined by a ***start*** integer and an ***end*** integer which are indexes into the *ColumnVisibility* expression character array. These two integers indicate a portion of the *ColumnVisibility* expression, beginning with ***start*** and up to but not including ***end***, that is encompassed by the subtree beginning with that *Node*. The ***type*** is an integer indicating whether the *Node* is the root of an AND or an OR subtree, or a TERM leaf *Node*. The child nodes are stored as a list of *Node* objects. Figure 4.2 is the parse tree for an example *ColumnVisibility* expression. Arrows in the tree indicate child *Nodes*.

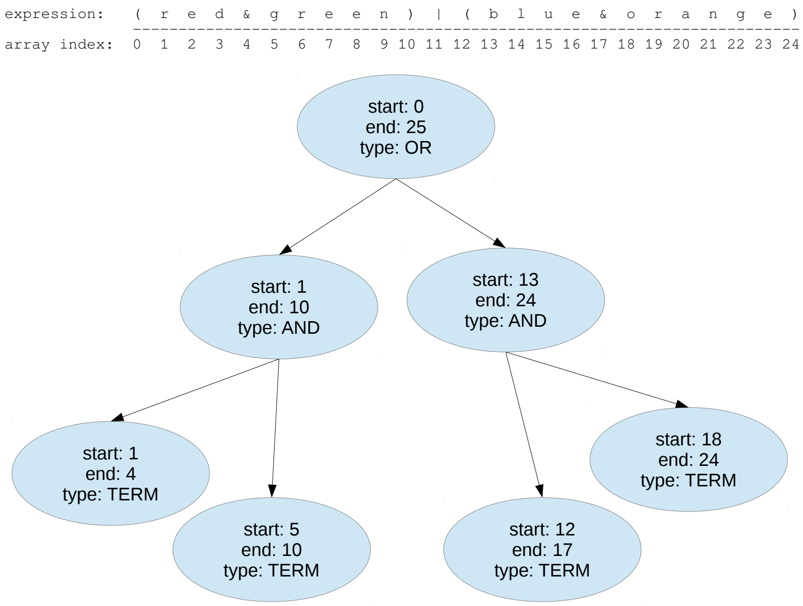

expression: (r e d & g r e e n) | (b l u e & o r a n g e)
array index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
start: 0 end: 25 type: OR
start: 1 end: 10 type: AND
start: 13 end: 24 type: AND
start: 1 end: 4 type: TERM
start: 5 end: 10 type: TERM
start: 12 end: 17 type: TERM
start: 18 end: 24 type: TERM

Figure 4.2: Example *ColumnVisibility* parse tree

4.2 Authorizations

Authorizations are security tokens provided by the client when querying the Accumulo database. Each token is a string that is intended to identify some level of data access authority. When a user is created in Accumulo, it is assigned a set of authorizations, stored in an *Authorizations* object. Any client that connects to Accumulo as that user, must submit a subset of the authorizations stored in the user account. The client may choose which of the authorizations are necessary for each query. Any query that is submitted with authorizations outside of the set stored in the user account will fail. If the client provides an appropriate subset of authorizations, the provided authorizations are compared to the *ColumnVisibility* expression associated with each key-value pair in the requested range of entries. If the authorizations satisfy the *ColumnVisibility* expression, that key-value pair is returned to the client.

For the remainder of this chapter, we provide a detailed guide through the Accumulo code that processes authorizations during data queries. We use Accumulo version 1.5.0 as the reference source code [34]. The discussion is divided into three parts. First, the client code determines the appropriate authorizations and sends them with the data query to the tablet server. Next, the tablet server receives the query request from the client and retrieves the appropriate key value pairs. Finally, we discuss the policy enforcement point at which the tablet server filters the results that are returned to the client. Filtering is based on a comparison of the client authorizations and the column visibility associated with each key-value pair.

Throughout this discussion, we reference three Java constructs: objects, fields, and methods. Objects are italicized for clarity. Fields, or variables within objects, are further differentiated using bold font. Methods, or object functions, are also bold but have a set of parentheses at the end of the name. The first time we reference each construct, we present the full package name to establish its location within the source code. Subsequent references include only that portion of the name necessary to avoid ambiguity. We do not cover all of the Accumulo code used to process queries, and the arguments noted for each step are not necessarily all the arguments required to properly execute that portion of code. These omissions allow us to focus on authorization processing within the query framework.

4.2.1 Client Authorization Handling

To query data, Accumulo client applications must first connect to an Accumulo instance using valid user credentials. Using a set of authorizations, the client creates a scanner that utilizes that connection to retrieve the appropriate data. The scanner provides an iterator framework that the client uses to step through the results of the query. The iterator sends the query request to the appropriate tablet server and supplies the results to the client. Figure 4.3 shows the flow of authorizations through the client code.

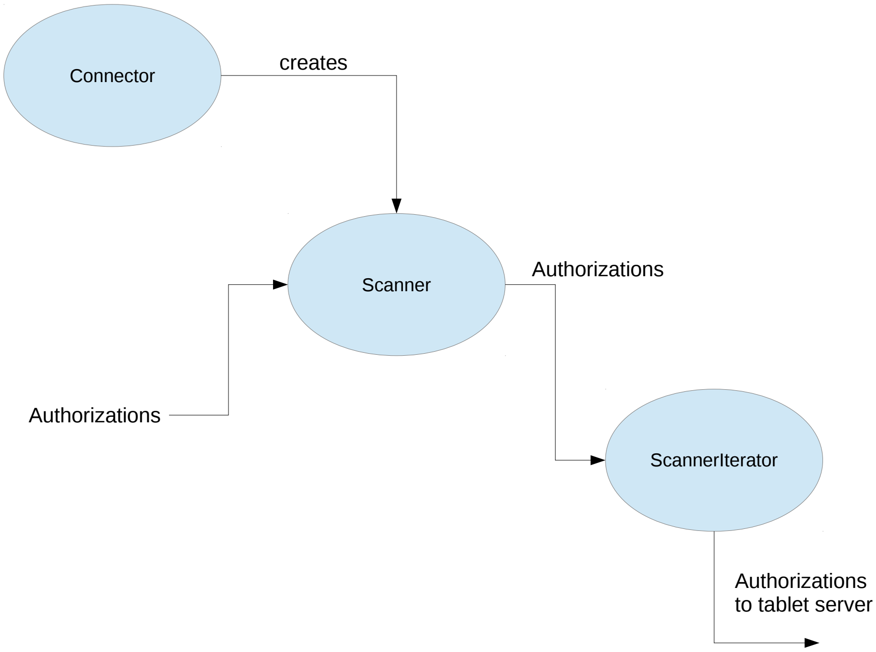


Figure 4.3: Client side *Authorizations* flow

Connect to Accumulo instance

- The client connects to an Accumulo instance by instantiating *accumulo.core.client.Connector* using the appropriate username and password
- The code implementing a *Connector* object is supplied in *accumulo.core.client.impl.ConnectorImpl*

Create a scanner

- The client obtains authorizations directly from user or from 3rd party authentication service
- The client instantiates *accumulo.core.security.Authorizations* using user authorization strings
- The client calls *Connector.**createScanner()*** with *Authorizations* as an argument
- ***createScanner()*** instantiates *accumulo.core.client.Scanner*
- *Scanner* implementation code is supplied in *accumulo.core.client.impl.ScannerImpl*

Scanner retrieves results from tablet server

- When the client iterates through the *Scanner* results, the *ScannerImpl.**iterator()*** method is called
- ***iterator()*** uses *Authorizations* to instantiate a *accumulo.core.client.impl.ScannerIterator*
- *ScannerIterator* constructor uses *Authorizations* to instantiate *accumulo.core.client.impl.ThriftScanner.ScanState*
- *ScannerIterator.**run()*** calls *ThriftScanner.**scan()*** with *ScanState* as an argument
- ***scan()*** calls *accumulo.core.tabletserver.thrift.TabletClientService.Client.**startScan()*** with *Authorizations* as an argument
- ***startScan()*** calls *Client.**send_startScan()*** with *Authorizations* as an argument
- ***send_startScan()*** instantiates *TabletClientService.startScan_args* and stores *Authorizations* in *startScan_args.**authorizations***
- ***send_startScan()*** calls *Client.**sendBase()*** with the string “startScan" and *startScan_args* as arguments
- ***sendBase()*** implementation code is supplied in *thrift.TServiceClient.**sendBase()***
- ***sendBase()*** sends “startScan" to tablet server then calls *startScan_args.**write()***
- ***write()*** calls *startScan_args.startScan_argsStandardScheme.**write()*** with *startScan_args* as an argument
- ***write()*** sends each argument from *startScan_args* to the tablet server sequentially
- *Client.**startScan()*** calls *Client.**recv_startScan()*** to get results from tablet server

4.2.2 Tablet Server Authorization Handling

The tablet server receives the query request, including authorizations, from the client. The tablet server first checks the authorizations against those stored in the user account. If the authorizations are a subset of the user account authorizations, the tablet server creates an iterator to scan the appropriate tablet or tablets. The iterator filters results based on comparison of authorizations and column visibilities. Figure 4.4 illustrates server side authorization flow.

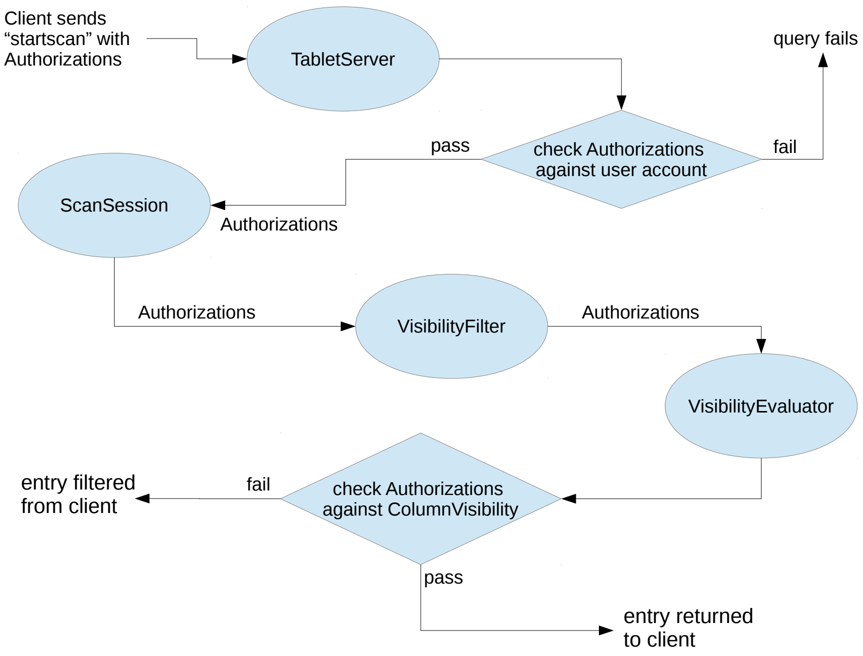


Figure 4.4: Server side *Authorizations* flow

Start tablet server

- *accumulo.start.Main* starts an *accumulo.server.tabletserver.TabletServer* process
- *TabletServer.**main()*** calls *TabletServer.**run()***
- *TabletServer.**run()*** calls *TabletServer.**startTabletClientService()***
- ***startTabletClientService()***
  - instantiates a *TabletServer.ThriftClientHandler* object
  - uses *ThriftClientHandler* to instantiate an *accumulo.core.tabletserver.thrift.TabletClientService.Iface* object
  - uses *Iface* to instantiate a *TabletClientService.Processor* object
  - calls *TabletServer.**startServer()*** with *Processor* as an argument

Initialize scan

- The *TabletServer* receives the client query request with “startScan" method indicated
- *Processor* maps “startScan" string to call to *ThriftClientHandler.**startScan()*** method
- *TabletServer* executes ***startScan()*** with client *Authorizations* as an argument
- ***startScan()***

  - calls *accumulo.server.security.SecurityOperation.**getUserAuthorizations()*** with client user credentials as an argument
  - verifies that *Authorizations* are a subset of the authorizations listed in the client's Accumulo user account
  - calls ***onlineTablets.get()*** to locate the appropriate *accumulo.server.tabletserver.Tablet* to fulfill the client request
  - instantiates a *TabletServer.ScanSession* and stores *Authorizations* in *ScanSession.**auths***
  - instantiates a *Tablet.Scanner* by calling *Tablet.**createScanner()*** with *Authorizations* as an argument
  - stores *Scanner* in *ScanSession.**scanner***
  - calls *ThriftClientHandler.**continueScan()*** with *ScanSession* as an argument

Iterate through requested data

- ***continueScan()*** calls *accumulo.server.tabletserver .TabletServerResourceManager.**executeReadAhead()*** with *ScanSession.NextBatchTask* as an argument
- ***executeReadAhead()*** calls *NextBatchTask.**run()***
- ***run()*** calls *ScanSession.Scanner.**read()***
- ***read()***
  - instantiates *Tablet.ScanDataSource* with *Authorizations* as an argument
  - uses *ScanDataSource* to instantiate *accumulo.core.iterators.system.SourceSwitchingIterator*
  - calls *Tablet.**nextBatch()*** with *SourceSwitchingIterator* as an argument
- ***nextBatch()*** calls *SourceSwitchingIterator.**seek()***
- *SourceSwitchingIterator.**seek()*** calls *ScanDataSource.**createIterator()***
- ***createIterator()*** uses *Authorizations* to instantiate *accumulo.core.iterators.system.VisibilityFilter*
- *VisibilityFilter* constructor uses *Authorizations* to instantiate *accumulo.core.security.VisibilityEvaluator*
- *SourceSwitchingIterator.**seek()*** calls *ScanDataSource.**readNext()***
- ***readNext()*** calls *VisibilityFilter.**seek()***
- *VisibilityFilter.**seek()*** implementation code is supplied in *accumulo.core.iterators.Filter.**seek()***

- *Filter.**seek()*** calls *Filter.**findTop()***

Check visibility of each key-value pair

- *Filter.**findTop()*** calls *VisibilityFilter.**accept()*** with the key and value as arguments
- ***accept()*** calls *VisibilityEvaluator.**evaluate()*** with *accumulo.core.security.ColumnVisibility* taken from the key as an argument
- ***evaluate()*** verifies that the *Authorizations* satisfy the *ColumnVisibility* expression

4.2.3 Checking Authorizations against Visibilities

The policy enforcement point of Accumulo's cell-level security is the comparison of client supplied *Authorizations* against the *ColumnVisibility* expressions in each key-value pair. At this point, Accumulo decides whether to return data to the user. The Accumulo construct that performs the comparison is the *VisibilityEvaluator*. The *VisibilityEvaluator* uses the parse tree constructed for the *ColumnVisibility* expression and evaluates it against the *Authorizations*. The *VisibilityEvaluator* starts at the root of the *ColumnVisibility* parse tree and works toward the leaves. It checks the type of each *Node* in the tree to determine if it is a leaf *Node*. If the *Node* is a leaf *Node*, the *VisibilityEvaluator* checks whether the authorization token associated with that *Node* is present in the *Authorizations* provided by the client. If the *Node* is not a leaf *Node*, the *VisibilityEvaluator* evaluates the *Node*'s children. Accumulo will not return data to the client unless the client supplied *Authorizations* satisfy the entire boolean expression described by the *ColumnVisibility* parse tree.

The evaluation algorithm is performed by the *VisibilityEvaluator.**evaluate()*** method. The *Authorizations* are stored as a field of the *VisibilityEvaluator* object. The algorithm begins by examining the root *Node*. If the root *Node* is a TERM, ***evaluate()*** returns the result of *Authorizations.**contains(term)***. If the root *Node* is an AND or an OR *Node*, ***evaluate()*** is called recursively on the child *Nodes*. An AND *Node* will return TRUE if both of its children return TRUE. An OR *Node* will return TRUE if any of its children return TRUE. ***evaluate()*** returns TRUE if the *Authorizations* satisfy the full *ColumnVisibility* expression, otherwise it returns FALSE. Pseudocode for the ***evaluate()*** method is shown in Figure 4.5.

```
Boolean evaluate(Node) {

    if Node.type == TERM:
        return Authorizations.contains(Node.term)

    if Node.type == AND:
        for child in Node.children:
            if !evaluate(child) return FALSE
        return TRUE

    if Node.type == OR:
        for child in Node.children:
            if evaluate(child) return TRUE
        return FALSE
}
```

Figure 4.5: Pseudocode for *evaluate()* algorithm

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Accumulo Client Applications

Accumulo provides a client API that allows applications to programmatically interact with Accumulo data. Client applications typically add more data management features and analysis capabilities such as a graphical user interface to browse data, a more expressive query language, data processing libraries for interpreting raw data, or graphical output for simpler consumption by end-users. This chapter introduces interaction between Accumulo and client applications and discusses some representative example applications using Accumulo.

5.1 Key Accumulo Client Interfaces

There are some Accumulo interfaces that are commonly used by client applications independent of implementation [29]. These interfaces allow Accumulo clients to connect to an Accumulo instance, write data to tables in Accumulo, and retrieve specific data entries from Accumulo.

Connector. To connect to an Accumulo instance, the client creates a *Connector* object. The *Connector* is constructed based on the location of the Accumulo master, and the credentials for the user on behalf of whom the client application is operating. The *Connector* establishes the line of communication between the client and Accumulo.

BatchWriter. Once connected to Accumulo, the client writes data using a *BatchWriter* object. The *BatchWriter* is constructed using the name of the destination table. The elements of the key-value pair are stored in a *Mutation* object which the *BatchWriter* sends to Accumulo.

Scanner. To retrieve data, the client uses a *Scanner* object. A *Scanner* is constructed using the name of the table, the authorization tokens used to access the data, and the range of data requested. The *Scanner* provides an iterator that the client uses to step through the results of the scan.

These interfaces form the foundation of client interaction with Accumulo and allow client applications to perform basic write and read operations to store and retrieve data. Figure 5.1

is a sample of code demonstrating a client connecting to Accumulo, storing an entry, then retrieving that entry. This example assumes that the user *username* and table *tableName* have already been established in the Accumulo instance *instanceName*.

```
//Connect to Accumulo instance
Instance instance = new ZooKeeperInstance("instanceName","zooServerName");
Connector conn = instance.getConnector("username",new PasswordToken("password"));

//Create entry
Mutation mutation = new Mutation("rowName");            //row id
mutation.put("columnFamilyName",                        //column family
             "columnQualifierName",                     //column qualifier
             new ColumnVisibility("visibilityName"),    //column visibility
             System.currentTimeMillis(),                //timestamp
             "entryValue");                             //value

//Store entry in Accumulo
BatchWriter writer = conn.createBatchWriter("tableName",new BatchWriterConfig());
writer.addMutation(mutation);
writer.close();

//Retrieve entry
Scanner scan = conn.createScanner("tableName",new Authorizations("visibilityName"));
for (Entry<Key,Value> entry : scan) {
    System.out.println(entry.getValue().toString());
}
```

Figure 5.1: Accumulo client code example

5.2 Multi-User Client Applications

When using Accumulo as part of a data management system, the client application will likely have many users that need to access different subsets of data. Accumulo provides cell-level security labeling to facilitate data segregation in a multi-user environment. Client applications, however, are not intended to run under the identity of different users or to authenticate to Accumulo under the identities of different users [29]. Instead, one Accumulo user is created and the client application accesses Accumulo through that user's credentials. The client application access control policy must include a strategy for associating the appropriate set a privileges with each user.

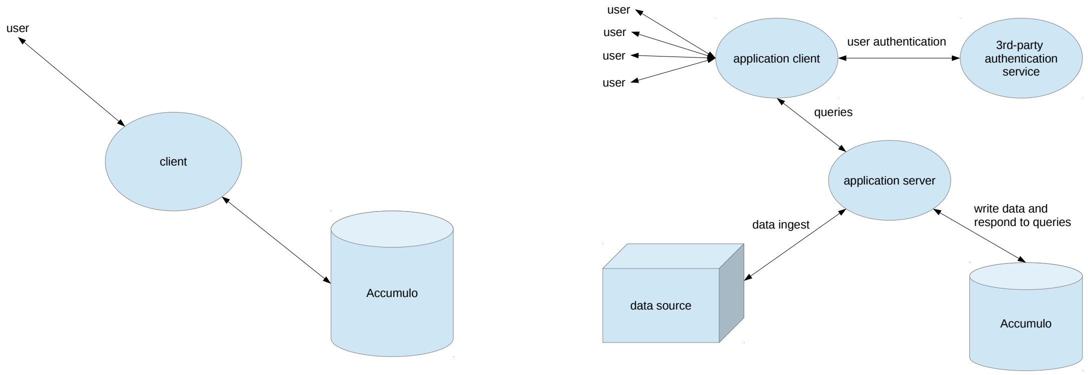
There are two sets of privileges client applications manage. Administrative permissions include user management, system settings, and the ability to access or modify tables. Cell-level authorizations allow users to access Accumulo entries.

Accumulo provides no assistance to the client in managing administrative permissions associated with application users. The Accumulo user necessarily has *all* administrative permissions required for the client application to function on behalf of any application user. Thus, an application user has all of the same permissions as the Accumulo user, unless the client takes steps to restrict user activity.

Controlling user access to data is a distinct problem from managing administrative permissions. The ability to access individual entries in Accumulo is dependent on the *Authorizations* provided by the user. The Accumulo user is assigned *Authorizations* covering the entire set of *Authorizations* any application user might need. It is the responsibility of the application to verify user credentials and prepare an appropriate set of *Authorizations* that reflect the user's permissions prior to querying Accumulo. Accumulo's cell-level security integrates security label processing into the database, but the client application must have a reliable procedure in place for verifying the identity of its users and associating appropriate *Authorizations* with each query submitted by a user.

5.3 Accumulo Client Examples

Accumulo clients can be simple applications that use only the native Accumulo client API (Figure 5.2(a)), or they can scale to much larger applications that provide a more abstract user interface and integrate with other applications (Figure 5.2(b)). We provide three examples of Accumulo client applications that illustrate the range of potential use cases.



(a) Simple Accumulo Client

(b) Non-trivial Accumulo client

Figure 5.2: Examples of Accumulo client structure

5.3.1 Trendulo

Trendulo [35] is a demonstration application developed for Accumulo. It is composed of an ingest application to store Twitter data in Accumulo and a web application that allows users to query the Twitter data. The ingest application is written in Java and interfaces with the Twitter Streaming API. The web application is written in HTML, JavaScript, and Java and incorporates several open source application development tools including Spring, JQuery, Bootstrap, ICanHaz, and Highcharts. It can be deployed using an open source web server such as Apache httpd or Nginx. Web application users issue simple queries to view Twitter trend data, showing frequency of target keywords over various time periods. Trendulo does not use column visibilities and does not differentiate between individual users. As a result, Trendulo provides little insight into the data security features of Accumulo.

5.3.2 Sqrrl

Sqrrl is a company founded by Accumulo developers that provides a large-scale enterprise data management solution. Sqrrl Enterprise [36] uses Accumulo to facilitate real-time application development. It provides its own methods for streaming data ingest and for batch ingest of static data (i.e., of JSON or CSV data). It also implements security controls that incorporate Accumulo's cell level security. It can identify and authenticate users, provide automatic data labeling based on organizational policy, and provide data encryption for an additional level of security. Sqrrl Enterprise also enables complex data analysis through additional data models, more expressive query languages, an indexing framework, and custom iterators. Sqrrl Enterprise is an example of a production application that implements user management policies; however, lack of access to this application prevents us from analyzing it as a case study.

5.3.3 Koverse

Koverse [37] is a data storage and analysis framework that is focused on operationalizing large amounts of data. Koverse automatically processes data upon ingest to store it in a consumable form. It uses role-based access control to manage multiple users but relies on third party applications to make use of Accumulo's cell-level security. Koverse provides data analysis algorithms that can merge data sets and identify meaningful relationships within large data sets. Koverse also provides support for developers to extend Koverse capabilities into custom applications. Koverse is the data query interface for the Naval Tactical Cloud project [10]. In the next chapter, we examine Koverse as a case study in how Accumulo is integrated into a production application.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6:
Accumulo Client Case Study

Koverse is a large-scale data management application that uses Accumulo for persistent data storage. In this chapter, we examine Koverse as a case study of Accumulo client application design. We do not provide a complete description of Koverse functionality. Instead, we focus on components of Koverse that illustrate interaction with Accumulo. We describe user management in a multi-user environment with a focus on data access authorization. Additionally, we illustrate the process of executing queries in Koverse to include the transformation of a user query into an Accumulo *Scanner*. These core functions form the foundation of Accumulo client design.

6.1 Architecture

The Koverse application has two main components—the Koverse server and the Koverse web application. The web application is the front end for Koverse and provides a graphical user interface. It is written in HTML, JavaScript, and Java and uses the JBoss development framework [38]. Built in applications within the web interface allow users to:

- Manage data collections
- Import data from external sources
- Query data
- Analyze data through transforms
- Manage users and groups

Users can perform all Koverse functionality through the web interface, but it is also possible to interact directly with the Koverse server using the Koverse API.

The Koverse server processes requests from the Koverse web application. It is written in Java, and interacts with the Accumulo client API. The Koverse server also interacts with third-party applications to perform authentication and security token assignment for Koverse users. Figure 6.1 illustrates component interaction in a Koverse environment.

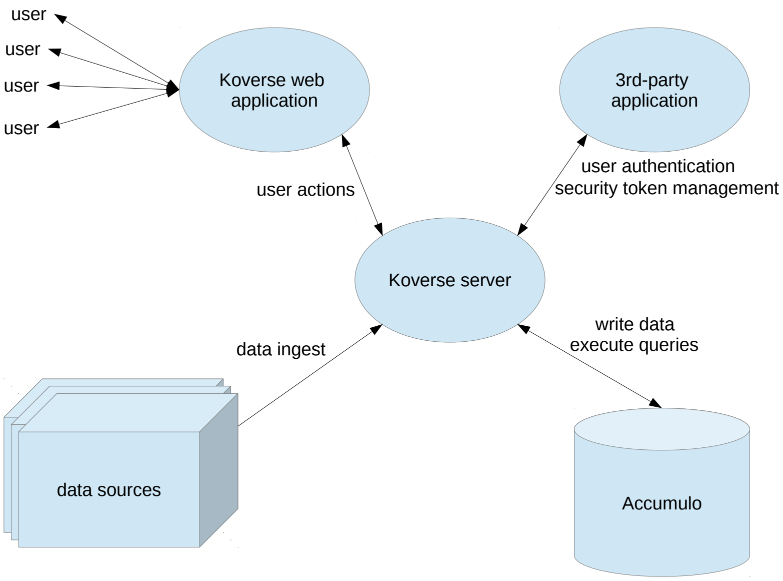


Figure 6.1: Koverse application architecture

6.2 Data Model

Koverse overlays its own data model on top of the Accumulo data model. Koverse stores data in *Records* which are sets of key-value pairs, referred to as *Fields*. The *Field* values can be one of several native data types (e.g., strings, integers, floating point numbers, timestamps, geospatial data). *Field* names must be unique within a *Record*, but may be reused across *Records*. *Fields* are not strongly typed, so two different *Records* using the same *Field* name may have different types associated with those *Fields*. *Records* can provide more complicated structure to data by nesting additional *Fields* within a *Field* value. Koverse applies security labels, analogous to Accumulo *ColumnVisibility* objects, at the *Record* level. When a *Record* is written to Accumulo, each *Field* is mapped into Accumulo's column-oriented model (see Table 6.1).

Each Koverse *Record* is assigned to one *Collection*. A *Collection* forms a set of related *Records*. *Collections* are schema-less, and each *Record* in the *Collection* can have a unique *Field* structure, including the number of *Fields* and the types of each *Field*. Mappings of *Records* to *Collections* and user *Collection* permissions are stored in a Java Persistence API (JPA) [39] system that is separate from Accumulo. No information indicating the *Collection* corresponding to a *Record* is stored in Accumulo.

A *Collection* can be thought of as a table in a relational database where each *Record* is a row and the *Fields* are columns. *Collections*, however, do not map to Accumulo tables. Koverse stores all data in two distinguished tables in Accumulo: the *index* table and *record* table. The *index* table stores a portion of each *Record*, organized in a way that improves query execution. The *record* table stores all Koverse *Records* regardless of which *Collection* the *Record* is associated with. Each entry in the *record* table is one *Field* of a Koverse *Record*.

Accumulo entry	**Koverse *Record***
Row ID	Record ID
Column Family	not used
Column Qualifier	Field name
Column Visibility	Koverse security label
Timestamp	applied by Accumulo
Value	Field value

Table 6.1: Mapping a Koverse *Record* to an Accumulo entry

6.3 User and Group Management

Koverse can manage many concurrent users. Users are identified by a username and email address and authenticate using a password. Koverse administers role-based privileges through groups. Each group is given a set of privileges, and a user assumes the privileges of all the groups it is assigned to. User and group information is stored in the JPA system. Although there are many distinct Koverse users, the Koverse server accesses Accumulo through a single user (by default, the Accumulo root user). The Koverse server has the full set of permissions in Accumulo which includes reading, writing, and modifying any table or data entry. Controlling access to data stored in Accumulo is completely dependent upon Koverse's ability to associate appropriate privileges with the Accumulo requests it performs on behalf of users.

To authenticate, a Koverse user provides login credentials through the Koverse web interface. By default, Koverse manages user authentication locally. Koverse compares the user credentials against the stored login information for that user. If the credentials match, Koverse creates a session for the user. Koverse can also utilize a third-party authentication service. In that scenario, when the user submits credentials through the Koverse interface, Koverse forwards the credentials to the authentication service which verifies the user identity. Koverse verifies the response from the authentication service, and if the user provided appropriate credentials, creates a session for the user.

Once the user has authenticated, it must obtain authorization to perform tasks. For administrative tasks—such as user and group management, *Collection* configuration, and audit log access—Koverse checks the groups associated with the user. If any of the groups has permission to perform the requested task, the user is granted access.

To access data in a *Record*, users must have permission to access the *Collection* associated with that *Record*. During a query, Koverse checks the user's groups, and then checks if any of those groups have access to the *Collection*. If one of the user's groups has access to the *Collection*, the user is granted access to the *Collection*.

Access to a *Collection* does not guarantee that the user can access all *Records* in that *Collection*. If any of the requested *Records* have security labels, the user must provide an appropriate set of tokens to access those *Records*. Koverse does not natively manage the security tokens for each user. To obtain the necessary tokens, the user must authenticate to a third-party service. Koverse submits the user's credentials to the third party service which verifies the credentials and returns the appropriate tokens. Koverse stores the tokens for the duration of the user's session and uses them for any queries submitted by that user.

6.4 Queries

Users access data by submitting queries to the Koverse server. The Koverse web interface has built in search functionality that allows users to query data in a way that resembles an Internet search engine. Users can search for a term in any *Field*, specify a value for a *Field*, or search for a range of values in a particular *Field*. Koverse translates queries from the search application into a JavaScript Object Notation (JSON) [40] formatted list of *Field* names and values. Example search application queries with their respective JSON queries are shown in Figures 6.2 and 6.3 [41].

```
mary
"mary had a"
name:mary
name:mary occupation:shepherd
height:[60 TO 70]
```

Figure 6.2: Koverse Search application query examples, from [41].

```
{“$any”:“mary”}
{“$any”:“mary had a”}
{“name”:“mary”}
{“$and”:[{“name”:“mary”},{“occupation”:“shepherd”}]}
{“$and”:[{“height”:{“$gte:60”}},{“height”:{“$lte”:“70”}}]}
```

Figure 6.3: Koverse JSON query examples, after [41].

After parsing user queries and verifying their syntax, Koverse generates an internal representation of the query that resembles a SQL “SELECT” statement. The Koverse query is stored in a Java *SelectStatement* object and has the format:

```
SELECT(FieldNames,CollectionIDs,Expression,Offset,Limit)
```

The *CollectionIDs* and *FieldNames* restrict the search to specific *Collections* and specific *Fields*. The *Expression* is a restriction on the *Field* values and mirrors the submitted query. *Offset* and *Limit* allow the user to control the range of results that are returned. An *Offset* of *n* ignores the first *n* results, and a *Limit* of *m* returns a maximum of *m* results.

Once the query has been translated to a *SelectStatement*, it is executed in two stages. First an Accumulo *Scanner* is created to scan the *index* table to quickly locate the required *Records*. Koverse uses the results of the *index* scan to create another Accumulo *Scanner* for the *record* table. The range of the *record* table *Scanner* is set based on the results of the *index* table scan.

6.5 Tokens

Koverse has the ability to apply security labels at the *Record* level. When data is ingested in Koverse, the security label is applied as an Accumulo *ColumnVisibility* object (see Table 6.1). Although each *Field* in a Koverse *Record* is stored in a separate Accumulo column, Koverse maintains a *Record* as a single entity, and all Accumulo entries from the same *Record* are assigned the same *ColumnVisibility*. User queries for restricted *Records* must include a set of access tokens.

By default, security labels are not associated with any Koverse *Records*. If *Record* labels are desired, Koverse does not provide native support for user token management: this functionality requires interaction with a third party application. When the user authenticates, the third party application provides the proper set of tokens for that user. Those tokens are stored for the duration of the user's Koverse session. Koverse uses these tokens in any query the user makes and constructs the appropriate *Authorizations* object for the Accumulo *Scanner*.

CHAPTER 7: Information Security Discussion

Accumulo cell-level access control assists application developers with data access policy enforcement; however, it does not provide a complete information security solution. When describing Accumulo's security capabilities in a *PC World* interview, Accumulo developer Adam Fuchs noted, "[s]ince the applications in this model can push down the security model into the database and companion components, you don't have to solve that in the application" [42]. This statement, and similar ones from others in the Accumulo development community, may give developers a false sense of confidence in the level of security Accumulo can provide. Production applications must implement sound policy enforcement logic to integrate securely with Accumulo. In this chapter, we present potential security problems Accumulo client applications should consider. We do not provide an exhaustive list of all potential security concerns, but these examples should convince an application developer that information security is a significant problem that is not solved exclusively using native Accumulo functionality.

7.1 User and Privilege Management

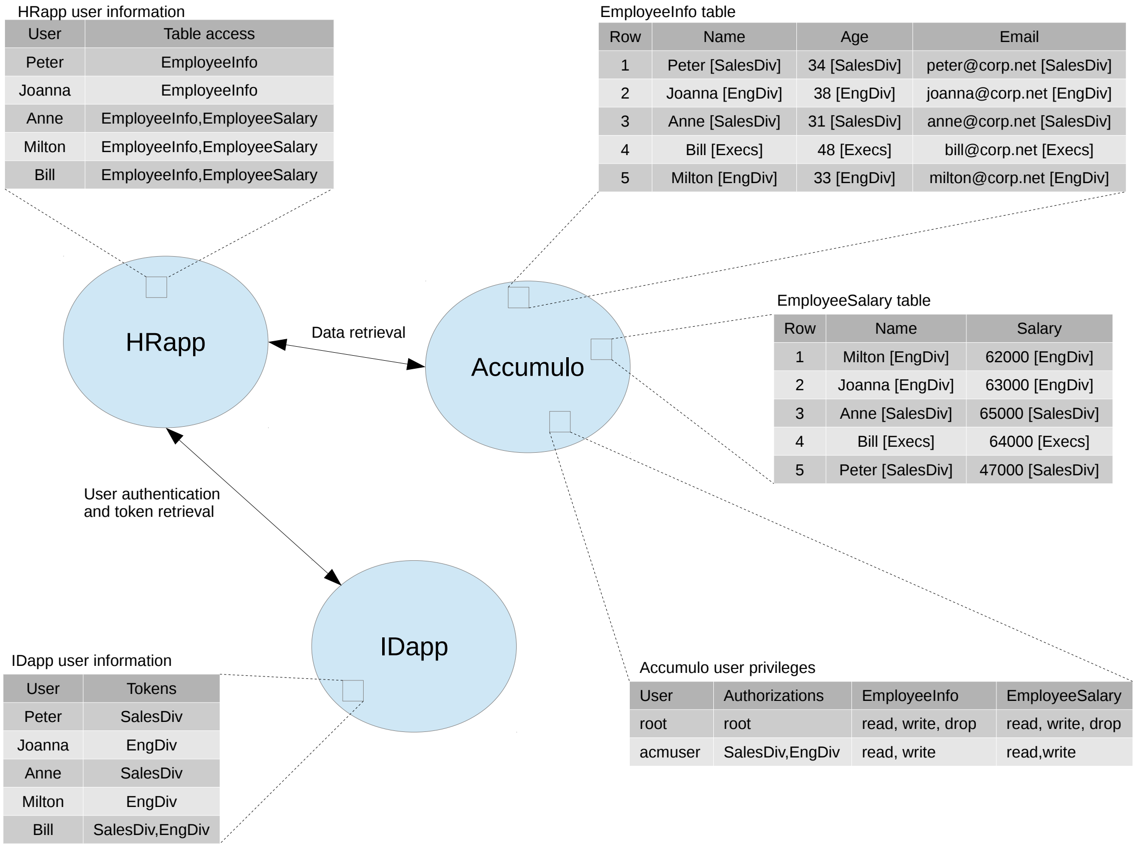
Proper management of user accounts and their associated privileges is critical for the security of any multi-user application. Functionality exists to manage users and privileges within Accumulo, but these interfaces are not likely to be used to manage client application users. As previously described, it is not expected that a large-scale Accumulo-based application will register a user account in Accumulo for every application user. Instead, Accumulo holds one user for the client application, which manages its own users separately. For clarity, in further discussion we refer to client application users as *appusers* and the Accumulo user as *acmuser*. Because many *appusers* are mapped onto one *acmuser*, there is no ability to differentiate between *appusers* at the Accumulo level. The client application authenticates to Accumulo as the *acmuser*, but must authenticate *appusers* and assign appropriate privileges prior to making any Accumulo requests.

There are several types of privileges to consider in an Accumulo application. System permissions give users the capability to perform administrative actions, such as creating or deleting users accounts and granting privileges to users. Table permissions allow users to modify table entries or table metadata. Cell-level authorizations control access to individual table entries. Each type of privilege may be present in Accumulo, managed separately by the client application, or both.

Because there are many *appusers* that access Accumulo through one *acmuser*, the *acmuser* must hold all privileges necessary to perform Accumulo operations on behalf of any *appuser*. It becomes the responsibility of the client application to prevent *appusers* from using inappropriate *acmuser* privileges. The fact that privileges at the application level do not necessarily map directly to privileges in Accumulo adds complexity to the problem. For instance, a complex data model at the application model may require a set of privileges that does not translate to Accumulo. In Koverse, data structures called *Collections* seem to map closely to Accumulo *Tables*. It may seem logical then for any privileges associated with a Koverse *Collection* to map to an Accumulo *Table*. Closer examination reveals that Koverse *Collections* do not directly parallel Accumulo *Tables*. In fact, there are two Accumulo *Tables* used to store data regardless of the number of *Collections* created in Koverse. Any privileges in Koverse associated with *Collections* management have no direct meaning in Accumulo.

Cell-level *Authorizations* map more closely from the application to Accumulo, but even these privileges may not translate directly. In the Koverse data model, security labels are applied at the *Record* level. When the *Record* is inserted into Accumulo, the *Record* is split into many entries and the security label is modified prior to being applied to all entries from the *Record*. To manage *Record* level access control, Koverse utilizes a third-party service that provides a set of access tokens for each user. In the application, the *appuser* accesses a *Record* using these tokens, but at the Accumulo level, the *acmuser* accesses multiple entries with a set of *Authorizations* that are distinct from the tokens provided by the third-party service.

The following example illustrates a user management scenario and highlights potential complications. Consider a fictional enterprise human resource information application, *HRapp*, illustrated in Figure 7.1. *HRapp* stores employee information in two Accumulo *Tables*—*EmployeeInfo* and *EmployeeSalary*—to isolate sensitive salary information from more general personal information. The *Tables* are shown in a relational table format for illustrative purposes. To understand the mapping to Accumulo entries, consider the entry for Peter's age in the *EmployeeInfo Table*. The entry in Accumulo would have the following structure: *Row*='1' *ColumnFamily*='' *ColumnQualifier*='Age' *ColumnVisibility*='SalesDiv' *Value*='34'. *HRapp* logic manages user access to each *Table*.



HRapp user information

User	Table access
Peter	EmployeeInfo
Joanna	EmployeeInfo
Anne	EmployeeInfo,EmployeeSalary
Milton	EmployeeInfo,EmployeeSalary
Bill	EmployeeInfo,EmployeeSalary

EmployeeInfo table

Row	Name	Age	Email
1	Peter [SalesDiv]	34 [SalesDiv]	peter@corp.net [SalesDiv]
2	Joanna [EngDiv]	38 [EngDiv]	joanna@corp.net [EngDiv]
3	Anne [SalesDiv]	31 [SalesDiv]	anne@corp.net [SalesDiv]
4	Bill [Execs]	48 [Execs]	bill@corp.net [Execs]
5	Milton [EngDiv]	33 [EngDiv]	milton@corp.net [EngDiv]

EmployeeSalary table

Row	Name	Salary
1	Milton [EngDiv]	62000 [EngDiv]
2	Joanna [EngDiv]	63000 [EngDiv]
3	Anne [SalesDiv]	65000 [SalesDiv]
4	Bill [Execs]	64000 [Execs]
5	Peter [SalesDiv]	47000 [SalesDiv]

IDapp user information

User	Tokens
Peter	SalesDiv
Joanna	EngDiv
Anne	SalesDiv
Milton	EngDiv
Bill	SalesDiv,EngDiv

Accumulo user privileges

User	Authorizations	EmployeeInfo	EmployeeSalary
root	root	read, write, drop	read, write, drop
acmuser	SalesDiv,EngDiv	read, write	read,write

Figure 7.1: HRapp example application.

HRapp authenticates its users using a third-party service called *IDapp* that verifies user credentials and returns an appropriate set of access tokens. *HRapp* interacts with Accumulo through a single user, *acmuser*, which has full access to all data. Accumulo *ColumnVisibilities* are applied based on the responsible organizational division—*EngDiv* for Engineering employees, *SalesDiv* for Sales employees, and *Execs* for Executives. In each division there is an operational manager and an human resources manager. The operational manager has access to general employee information for his division, and the human resources manager has access to both general and salary information for her division. Executives have access to all employee information.

When users log into *HRapp*, they provide a set of credentials. *HRapp* forwards these credentials to *IDapp* which verifies the user identity and returns the user's access tokens. *HRapp* stores tokens for the duration of the user's session. When a user issues a query, *HRapp* creates an Accumulo request containing the appropriate table name and tokens. *HRapp* does not allow users to query tables they do not have access to. For instance, if user *Peter*, who is an operational manager, queries the *EmployeeInfo Table*, he would receive the information in rows 1 and 3. If he queried the *EmployeeSalary Table*, his request would be denied. If user *Milton*, who is a human resources manager, queries the *EmployeeInfo Table*, he would receive rows 2 and 4. He would have to issue a separate query to retrieve information in the *EmployeeSalary Table* and would receive rows 1 and 2. User *Bill*, an executive, can query either *Table* and would receive all rows.

To enforce the above policy, HRapp must restrict user access to certain tables. The general application design—*HRapp* translates *appuser* requests into *acmuser* requests—essentially *requires* that table-level permissions be enforced by *HRapp* logic. The essential conflict stems from a combination of two application characteristics. First, the *acmuser* must have the ability to read all tables. Second, *Authorizations* do not specify table-level permissions in Accumulo. Thus, if *appuser* Johanna can cause *HRapp* to query the *EmployeeSalary* table, e.g., by misusing an interface, then the previously described access control policy will be violated. It is not enough, in this example, for *HRapp* to properly associate access tokens with each *appuser* via *IDapp* and rely on Accumulo to enforce all access control policy requirements. The ambient authority that allows *acmuser* to read all tables could be abused if *HRapp* fails to enforce table-level policies properly. In theory, table-level enforcement could be pushed to Accumulo, if *appuser* access tokens were table specific. Following our example, correcting this problem would require an additional term added to the *ColumnVisibilities* in the *EmployeeSalary Table* (e.g., *[EngDiv&Salary]*) and updating the appropriate user tokens in *IDapp*. This adds an additional layer of administration and complexity when adding new application users or new database entries.

7.2 NoSQL Injection

Injection attacks are a common exploitation vector, particularly in web applications. They are commonly used to retrieve sensitive or restricted data from application databases and have been identified as a significant information security concern [43]. Injection attacks typically occur when an application accepts user input insecurely. Attackers can craft input in such a way that forces the application server to perform actions that are not meant to be available to normal users. Injection attacks can allow attackers to perform any action of their choosing on the database, including reading, writing, inserting or deleting arbitrary data.

Injections attacks against SQL databases have been well explored, but similar attacks have also been reported in the expanding NoSQL database community. The OWASP organization proposes that NoSQL injection attacks may have more significant impact than SQL injection because they are executed in a lower level procedural API [44]. Table 7.1 summarizes potential vulnerabilities for some common NoSQL databases. We leave rigorous examination of the applicability of these attacks to Accumulo as an open problem, but it is important to note that divorcing an application from SQL databases does not remove the potential for injection attacks.

Name	Type	Interface Languages	Documented Query-Language Attacks
Cassandra	column	CQL, drivers available for Java, C#, Python	manual construction of query strings [45]
MongoDB	document	JavaScript, drivers available for many common languages	“$where” attacks [46]–[48]
Redis	key/value	standard map manipulation commands (e.g., GET, SET), drivers available for many common languages	redisCommand() attack [49]
CouchDB	document	HTTP, JavaScript	JavaScript injection, file system traversal, XSS [50]–[52]
Tokyo Cabinet	key/value	C, Perl, Ruby, Java, Lua	binary protocol injection vulnerabilities [53]

Table 7.1: Summary of NoSQL stores and documented query language vulnerabilities.

7.3 Information Security Policy Enforcement

Terminology used in the Accumulo development community may give a false impression of Accumulo’s security policy enforcement capability. Descriptions of Accumulo frequently contain terms and phrases that are typically associated with Mandatory Access Control (MAC) policies, for example: “*mandatory* attribute-based *access control*” [28], access control through object “labels” [54], multiple “security levels” [29] or “security classifications” [10] stored together, and “intermingling data sets” [55]. According to the DOD Trusted Computer System Evaluation Criteria (TCSEC) standard, the only systems associated with labeling are class B1 and above, where those labels “shall be used as the basis for mandatory access control” [56]. This statement suggests that the use of data labeling is highly correlated with mandatory access control policies. The use of MAC terminology suggests that Accumulo can enforce an information flow control policy; however, Accumulo’s native functionality cannot enforce such a policy.

MAC is an access control and information flow control policy that uses labels to restrict access to objects based on a comparison of the subject and object security level. The

TCSEC standard states than in order to enforce a mandatory security policy, these labels must be applied to each object in the system and must reliably identify the the objects' sensitivity levels [56]. A MAC policy does not dictate access rules for individual subjects, but relies on labels to enforce access control. This stands in contrast to a Discretionary Access Control (DAC) policy that maintains a set of object access rights for each subject and allows subjects to grant and revoke access to other subjects for objects they own [57] .

An immediate indication of Accumulo's inability to enforce information flow policies is the absence of a lattice-based ordering of Accumulo labels. A key feature of a MAC policy is a lattice framework constructed by a partially ordered set of security levels [58]. The lattice is necessary for determining dominance between two different security levels. The dominance property determines if a subject is authorized to perform an action. Sandhu (1996) implements a mandatory policy using "role hierarchies" in a lattice framework [59]. In Accumulo, *ColumnVisibilities* are used to label data, but no mechanism exists for determining ordering of cell-level labels. Access to Accumulo entries is based on a byte-by-byte comparison of the boolean *ColumnVisibility* expression to user *Authorizations*. A client application would need to provided additional logic to determine ordering.

To further illustrate Accumulo's inability to enforce MAC, we consider the Bell-LaPadula [60] model, a well understood MAC policy. Bell-LaPadula identifies three properties that a secure system should exhibit. The simple security property requires that no user can read data with a higher classification than the user's security level. This property is commonly referred to as "no read up." The star property requires that no user can write data with a lower classification than the user's security level. This property is commonly referred to as "no write down." The tranquility property requires that no user can modify the classification level of data [57]. Accumulo's *ColumnVisibilities* may be able to enforce the simple security property. With proper assignment of *ColumnVisibilities* to data and *Authorizations* to users, Accumulo can ensure that users do not access unauthorized data (i.e. data for which the user does not hold the appropriate set of Authorizations). For instance, a user with a SECRET token would not be able to access TOP SECRET data; however, because Accumulo does not impose order on *ColumnVisibilities*, that user would also not be access UNCLASSIFIED data. To implement this ability, a user with SECRET clearance would need *Authorizations* that include both SECRET and UNCLASSIFIED tokens. This may or may not be a desired property in any particular implementation but illustrates a potential problem.

Accumulo does not enforce access controls on write operations in the same way as read operations. By default, there is no user *Authorizations* check when writing data. Any user can write data with any *ColumnVisibility* value. This may violate the star property. A user can read data with a high security label and write identical data to a cell with a lower label. This problem is also referred to as leakage in some literature. If the *Row*, *ColumnFamily*, and *ColumnQualifier* of the data are kept the same, this scenario would also violate the tranquility property. The old entry would be effectively re-labeled with a lower classification. Without restrictions administered by the client application, any user with write access can effectively downgrade the classification of any data.

Accumulo has an optional configuration setting that can be applied at the table level that prevents users from writing data with *ColumnVisibilities* that are not part of their *Authorizations* set. Recall that during read operations, Accumulo verifies that the subset of *Authorizations* provided in the query satisfies the *ColumnVisibility* associated with the requested data. Accumulo does not perform this check during write operations. Instead, Accumulo simply verifies that the appropriate *Authorizations* are associated with the user. In the recommended use case, in which all *appusers* operate through one *acmuser*, this check provides no protection. When the request reaches Accumulo, it is executed by *acmuser*, which holds the entire domain of *Authorizations* necessary for all *appusers*. Therefore, any *appuser* could write data with any *ColumnVisibility* within the domain. The constraint would, however, prevent *appusers* from writing data with nonsensical *ColumnVisibilities* in the context of the application.

Accumulo's loose restrictions on write operations prevent it from enforcing useful MAC properties. If a data access policy allows only read operations, Accumulo could be used to enforce the simple security property, but would likely require the client application to provide some additional logic to fully implement an ordered lattice framework, especially in the scenario in which multiple client application users are mapped to one Accumulo user. If client application users routinely write to the database, Accumulo could provide only DAC enforcement, and logic needed to enforce a MAC policy would have to be provided by the client application.

CHAPTER 8: Conclusion and Future Work

In this thesis, we studied Apache Accumulo's cell-level access control. This fine-grained access control can be used in data sets with varying degrees of sensitivity to maximize accessibility while maintaining the required level of secrecy. This security feature gives Accumulo a unique position in the quickly expanding NoSQL ecosystem and is particularly interesting for the DOD where it is being integrated into projects like the Naval Tactical Cloud.

8.1 Conclusions

We employed static analysis of source code to gain detailed insight into Accumulo's cell-level access control enforcement. We illustrated the execution path of a query starting at the client *Scanner* interface and ending at the enforcement point in the *TabletServer*. We formalized the syntax for a *ColumnVisibility* label and showed how *Authorizations* are compared to *ColumnVisibility* expressions to filter query results. These details provide more insight into Accumulo's security policy enforcement mechanisms that can be used for further study.

After understanding low-level details of Accumulo policy enforcement, we showed how Accumulo could be integrated into a larger application. We highlighted important interfaces in the client library needed to perform basic read and write operations. We identified several examples of applications that use Accumulo and detailed Koverse operation as a case study. We used Koverse to show how an application could develop a custom data model and map it to Accumulo. Most importantly, we showed how Accumulo's recommended user organization (multiple application users mapped to one Accumulo user) is implemented in practice. We showed how a custom application query can be translated to Accumulo queries. Although Koverse does not implement fine grained security by default, we showed how that functionality would interact with Accumulo if used. The Koverse case study gives readers a basic understanding of application integration with Accumulo. Our work can be interpreted as a first step toward a thorough analysis of Accumulo information security enforcement. Understanding the interaction between Koverse and Accumulo is particularly useful for readers who are concerned with how Accumulo may benefit security of sensitive DOD information.

We commented on potential security threats facing developers that build applications based on Accumulo. We used a hypothetical application to illustrate potential user management concerns. We identified injection attacks that have been carried out against other NoSQL databases and may be relevant to some uses of Accumulo. We commented on Accumulo's inability to enforce information flow policies. These examples serve to demonstrate that using Accumulo and it's cell-level security feature is not a full solution to access control problems unless Accumulo is paired with well-designed enforcement mechanisms in the client application. We believe that the combination of our technical discussion of Accumulo's cell-level access control enforcement, illustration of Accumulo integration in a larger application, and identification of potential security concerns may help future studies learn more about Accumulo information security and lead to development of more secure Accumulo based applications.

8.2 Future Work

The scope of this thesis was limited primarily to static analysis of Accumulo source code. We were able to provide a detailed description of Accumulo's security policy enforcement using this method, but there are other methods that could be used to further investigate information security in Accumulo. Potential areas for future research include:

Application vulnerability analysis

More detailed analysis could be done to determine if specific instantiations and configurations of Accumulo have any vulnerabilities that may lead to a security compromise. For instance, in Chapter 2 we list several known injection attacks against NoSQL databases, and follow-on studies could determine if these are applicable to Accumulo. A starting point for such studies could be an open source JSON interface for Accumulo called Jaccson [61]. According to its documentation, Jaccson's design is based on MongoDB's API, and therefore, may be susceptible to attacks similar to "$where" attacks used against MongoDB.

In addition to analysis of known attacks, future research could attempt to identify

Accumulo specific vulnerabilities using penetration testing tools such as OWASP Zed Attack Proxy or fuzzing tools. Many of these tools are protocol specific, so efforts could be made to adapt the general approach of a specific tool to testing of Accumulo or Accumulo based applications. Web applications are the most frequent targets for injection attacks and both Accumulo and HDFS supply web interfaces to monitor system performance. Koverse also provides a web interface and is a component of NTC. As Accumulo becomes more popular, there may be more large scale applications available for testing.

Network traffic analysis

Accumulo components reside on disjoint physical machines and must communicate across a network. Current versions of Accumulo communicate largely through remote procedure calls over TCP/IP via Apache Thrift's network stack [62]. If these communications are insecure, they could leak sensitive information. Future studies could analyze all network traffic generated by Accumulo components, determine what information is transmitted, and identify default communication security settings. Based on this traffic analysis, researchers could determine what information may be at risk and recommend vulnerability mitigation strategies.

Best practice configuration settings

The NoSQL ecosystem is relatively new and availability of security best practices is limited. Future work could include a survey of NoSQL databases to determine configuration properties that are security relevant. It may be possible to develop a general set of security related best practices for NoSQL systems, but the wide range of systems that fall under the NoSQL umbrella my require generalization to the point of triviality. In any case, the development of a set of best practices specific for Accumulo should be feasible.

Information flow control

We showed that Accumulo is not capable of enforcing information flow control policies without additional logic. Further research could propose how to achieve mandatory access control policy enforcement in an Accumulo application. One promising area of research is using the NSA's Cloud Security Gateway and Trusted Data Format to implement a integrity lock [63] style architecture. Another method could modify Accumulo to rely on a trusted operating system to enforce information flow policy following approaches explored by Nguyen *et al.* [64] and Roy *et al.* [65]. A successful study could validate the use of Accumulo in cross-domain DOD applications.

APPENDIX: Accumulo Installation

This guide covers the installation/configuration of Hadoop, Zookeeper and Accumulo. These instructions were tested using a fresh install of Ubuntu-12.04 LTS (64-bit).

Install Hadoop 1.2.1

This guide will install and configure a single node pseudo-distributed version of Hadoop.

1. Install Java

```
$ sudo apt-get install openjdk-6-jdk
$ java -version
java version "1.6.0_27"
OpenJDK Runtime Environment (IcedTea6 1.12.6) \
    (6b27-1.12.6-1ubuntu0.12.04.2)
OpenJDK 64-Bit Server VM (build 20.0-b12, mixed mode)
```

2. Disable ipv6 (recommended by many Hadoop users)

```
$ sudo vi /etc/sysctl.conf
```

   Add the following lines to the end:

```
#disable ipv6
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
net.ipv6.conf.lo.disable_ipv6 = 1
```

3. Download Hadoop 1.2.1 from one of the Apache mirrors[1], and unpack it.

```
$ wget http://goo.gl/OoR9TS -O hadoop-1.2.1.tar.gz
$ tar xzf hadoop-1.2.1.tar.gz
```

4. Define `JAVA_HOME` as the root of your Java installation.

```
$ vi hadoop-1.2.1/conf/hadoop-env.sh
```

[1] See http://hadoop.apache.org/releases.html

Adjust the following line:

```
export JAVA_HOME=/usr/lib/jvm/java-6-openjdk-amd64
```

5. Configure Hadoop Edit `hadoop-1.2.1/conf/core-site.xml` to reflect:

```
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://localhost:9000</value>
  </property>
</configuration>
```

Edit `hadoop-1.2.1/conf/hdfs-site.xml` to reflect:

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
  <property>
      <name>dfs.support.append</name>
      <value>true</value>
  </property>
</configuration>
```

Edit `hadoop-1.2.1/conf/mapred-site.xml` to reflect:

```
<configuration>
  <property>
    <name>mapred.job.tracker</name>
    <value>localhost:9001</value>
  </property>
</configuration>
```

6. Configure `ssh` to be passwordless. Test to see if a password is required, using the command:

```
$ ssh localhost
```

If you can’t ssh into localhost without a password execute the following:

```
$ ssh-keygen -t dsa -P ’’ -f ~/.ssh/id_dsa
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

Install Zookeeper 3.4.5

This guide will install and configure Zookeeper for standalone operation.

1. Download Zookeeper from one of the Apache mirrors[2], and unpack it.

```
$ wget http://goo.gl/1FQoec -O zookeeper-3.4.5.tar.gz
$ tar xzvf zookeeper-3.4.5.tar.gz
```

2. Create the configuration file `zookeeper-3.4.5/conf/zoo.cfg`:

```
tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
maxClientCnxns=100
```

The `dataDir` should point to an existing empty directory:

```
sudo mkdir /var/lib/zookeeper
sudo chown ‘whoami‘ /var/lib/zookeeper
```

Install Accumulo 1.5.0

This guide will install and configure Accumulo for a single computer.

1. Download Accumulo source from one of the Apache mirrors[3], and unpack it.

```
$ wget http://goo.gl/mG73aD -O accumulo-1.5.0-src.tar.gz
$ tar xzvf accumulo-1.5.0-src.tar.gz
```

2. Build Accumulo.

```
$ sudo apt-get install maven
$ cd accumulo-1.5.0
```

[2] See `http://zookeeper.apache.org/releases.html`

[3] See `http://accumulo.apache.org/downloads/`

```
$ mvn package -P assemble
$ cd ..
```

3. Copy configuration files to conf directory.

```
$ cp accumulo-1.5.0/conf/examples/512MB/native-standalone/* \
    accumulo-1.5.0/conf
```

4. Set `JAVA_HOME`, `HADOOP_HOME`, and `ZOOKEEPER_HOME`:

```
$ vi accumulo-1.5.0/conf/accumulo-env.sh
```

In particular, any lines featuring these exports should read:

```
export ZOOKEEPER_HOME=<your path>/zookeeper-3.4.5
export HADOOP_PREFIX=<your path>/hadoop-1.2.1
export JAVA_HOME=<your path, same as above for Hadoop>
test -z "$ACCUMULO_HOME"  && \
    export ACCUMULO_HOME=<your path>/accumulo-1.5.0
```

5. Create the directory indicated by the path variable `ACCUMULO_LOG_DIR`. This path is defined in the configuration script `accumulo-1.5.0/conf/accumulo-env.sh`. For example:

```
$ mkdir accumulo-1.5.0/logs
```

6. Accumulo requires the Hadoop "commons-io" java package. This is normally distributed with Hadoop. It should be located at `hadoop-1.2.1/lib/commons-io-2.1.jar`. If your Hadoop distribution does not provide this package, you will need to obtain it and put the "commons-io" jar file under `accumulo-1.5.0/lib`.

Starting Accumulo

Use the following steps to start the Accumulo instance, to verify installation.

1. Start Hadoop

```
$ hadoop-1.2.1/bin/hadoop namenode -format
$ hadoop-1.2.1/bin/start-all.sh
```

2. Verify Hadoop is running by browsing the following web interfaces. If you can connect to these pages, Hadoop is running:

   ```
   $ lynx http://localhost:50070/
   $ lynx http://localhost:50030/
   ```

3. Start Zookeeper

   ```
   $ zookeeper-3.4.5/bin/zkServer.sh start
   ```

4. Verify Zookeeper is running by connecting to the shell

   ```
   $ zookeeper-3.4.5/bin/zkCli.sh -server 127.0.0.1:2181
   ```

   You should a command prompt, like:

   ```
   [zk: 127.0.0.1:2181(CONNECTED) 0]
   ```

   To exit the shell, type ‘quit’.
5. Initialize Accumulo, to create an instance name and root password.

   ```
   $ accumulo-1.5.0/bin/accumulo init
   ```

6. Start Accumulo

   ```
   $ accumulo-1.5.0/bin/start-all.sh
   ```

7. Verify Accumulo running by browsing:

   ```
   $ lynx http://localhost:50095/
   ```

   Alternatively, verify Accumulo running by connecting to the shell:

   ```
   $ accumulo-1.5.0/bin/accumulo shell -u root
   ```

   Enter the root password you just created. You should see the prompt

   ```
   root@accumulo>
   ```

   Exit the shell by typing ‘quit’.

Stopping Accumulo

The following commands can be used to stop the running Accumulo instance:

```
$ accumulo-1.5.0/bin/stop-all.sh
$ zookeeper-3.4.5/bin/zkServer.sh stop
$ hadoop-1.2.1/bin/stop-all.sh
```

List of References

[1] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an architectural era: (It's time for a complete rewrite)," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, 2007, pp. 1150–1160.

[2] D. Harris. (2013, June 7). Under the covers of the NSA's big data effort. *Gigaom*. [Online]. Available: http://gigaom.com/2013/06/07/under-the-covers-of-the-nsas-big-data-effort/

[3] C. Young. (2012, Mar. 12). Military intelligence redefined: Big data in the battlefield. *Forbes*. [Online]. Available: http://www.forbes.com/sites/techonomy/2012/03/12/military-intelligence-redefined-big-data-in-the-battlefield/

[4] G. Gardner. (2014, Mar. 13). Enabling battlefield big data 'on the move'. *Defense Systems*. [Online]. Available: http://defensesystems.com/articles/2014/03/13/commentary-gardner-battlefield-big-data.aspx

[5] J. Edwards. (2014, June 2). Military, intel turn to big data for better situational awareness. *FederalTimes*. [Online]. Available: http://www.federaltimes.com/article/20140602/FEDIT/306020009/Military-intel-turn-big-data-better-situational-awareness

[6] C. Wood. (2014, Jan. 6). How does the military use big data. *Emergency Management*. [Online]. Available: http://www.emergencymgmt.com/safety/Military-Use-Big-Data.html

[7] R. Locker. (2014, Feb. 21). Pentagon seeks 'big code' for 'big data'. *MilitaryTimes*. [Online]. Available: http://www.militarytimes.com/article/20140221/NEWS04/302210019/Pentagon-seeks-big-code-big-data-

[8] Information Innovation Office. (n.d.). Mining and Understanding Software Enclaves (MUSE). *DARPA*. [Online]. Available: http://www.darpa.mil/Our_Work/I2O/Programs/Mining_and_Understanding_Software_Enclaves_(MUSE).aspx

[9] Information Innovation Office. (n.d.). XDATA. *DARPA*. [Online]. Available: http://www.darpa.mil/Our_Work/I2O/Programs/XDATA.aspx

[10] "Achieving a naval data strategy: Leveraging the Unified Cloud Data (UCD) ecosystem as the pathfinder for a Naval data ecosystem," White Paper, Office of Naval Research, Code 31 and U.S. Navy TENCAP, Feb. 2014.

[11] Couchbase. (2012, Feb. 8). Couchbase survey shows accelerated adoption of NoSQL in 2012. [Online]. Available: http://www.couchbase.com/press-releases/couchbase-survey-shows-accelerated-adoption-nosql-2012

[12] C. Strauch, “NoSQL databases,” 2011, unpublished.

[13] A. Lakshman and P. Malik, “Cassandra: Structured storage system on a P2P network,” in *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, 2009, p. 5.

[14] E. Lai. (2008, Nov. 24). Google claims MapReduce sets data-sorting record, topping Yahoo, conventional databases. *Computerworld*. [Online]. Available: http://www.computerworld.com/s/article/9121278/Google_claims_MapReduce_sets_data_sorting_record_topping_Yahoo_conventional_databases

[15] E. A. Brewer, “Towards robust distributed systems,” presented at the Principles of Distributed Computing, Portland, OR, July 2000.

[16] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[17] A. Moniruzzaman and S. A. Hossain, “NoSQL database: New era of databases for big data analytics-classification, characteristics and comparison,” *International Journal of Database Theory & Application*, vol. 6, no. 4, 2013.

[18] R. Cattell, “Scalable SQL and NoSQL data stores,” *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2011.

[19] DB-Engines. (2014, Aug.). DB-Engines ranking. [Online]. Available: http://db-engines.com/en/ranking

[20] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, “Bigtable: A distributed storage system for structured data,” *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[21] E. Bertino and R. Sandhu, “Database security-concepts, approaches, and challenges,” *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 1, pp. 2–19, Jan.-Mar. 2005.

[22] L. Okman, N. Gal-Oz, Y. Gonen, E. Gudes, and J. Abramov, “Security issues in NoSQL databases,” in *2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2011, pp. 541–547.

[23] Cloud Security Alliance Big Data Working Group, “Expanded top ten big data security and privacy challenges,” White Paper, Apr. 2013. [Online]. Available: https://downloads.cloudsecurityalliance.org/initiatives/bdwg/Expanded_Top_Ten_Big_Data_Security_and_Privacy_Challenges.pdf

[24] C. Metz. (2012, July 17). NSA mimics Google, pisses off Senate. *Wired*. [Online]. Available: http://www.wired.com/2012/07/nsa-accumulo-google-bigtable

[25] J. Kepner, C. Anderson, W. Arcand, D. Bestor, B. Bergeron, C. Byun, M. Hubbell, P. Michaleas, J. Mullen, D. O’Gwynn *et al.*, “D4M 2.0 schema: A general purpose high performance schema for the Accumulo database,” in *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, 2013, pp. 1–6.

[26] S. M. Sawyer, B. David O’Gwynn, A. Tran, and T. Yu, “Understanding query performance in Accumulo,” in *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, 2013, pp. 1–6.

[27] R. Sen, A. Farris, and P. Guerra, “Benchmarking Apache Accumulo big data distributed table store using its continuous test suite,” in *2013 IEEE International Congress on Big Data*, 2013, pp. 334–341.

[28] A. Fuchs. (2012, July). Apache Accumulo. *National Security Agency*. [Online]. Available: http://www.oss-institute.org/storage/documents/Presentations/project_accumulo.pdf

[29] Apache Accumulo Project. (n.d.). Apache Accumulo user manual version 1.5. [Online]. Available: http://accumulo.apache.org/1.5/accumulo_user_manual.html

[30] Sqrrl, “Sqrrl enterprise cell-level security,” White Paper, n.d. [Online]. Available: http://sqrrl.com/media/Cell-Level-Security_FNL.pdf

[31] Oracle. (2014, Mar.). Configuring privilege and role authorization. [Online]. Available: http://docs.oracle.com/cd/E11882_01/network.112/e36292/authorization.htm

[32] Oracle. (2014, Mar.). Using Oracle virtual private database to control data access. [Online]. Available: http://docs.oracle.com/cd/B28359_01/network.111/b28531/vpd.htm

[33] J. Kelly. (2012, Aug. 20). Accumulo: Why the world needs another NoSQL database. *Devops Angle*. [Online]. Available: http://devopsangle.com/2012/08/20/accumulo-why-the-world-needs-another-nosql-database/

[34] The Apache Software Foundation. (2013, May). Apache Accumulo source code. [Online]. Available: http://archive.apache.org/dist/accumulo/1.5.0/accumulo-1.5.0-src.tar.gz

[35] J. Winick. (2012, Apr.). Trendulo. [Online]. Available: https://github.com/jaredwinick/Trendulo

[36] Sqrrl. (n.d.).“Sqrrl Enterprise: Unlock the power of big data,” White Paper. [Online]. Available: http://sqrrl.com/media/SQRRL_WP_Final.pdf

[37] Koverse. (2013, Sep.).“Big data and the data lake,” White Paper. [Online]. Available: http://koverse.com/whitepapers/Big-Data-and-the-Data-Lake-9_2013-.pdf

[38] RedHat. (n.d.). JBossDeveloper. [Online]. Available: http://www.jboss.org

[39] Oracle. (n.d.). Java Persistence API. [Online]. Available: http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html

[40] T. Bray. (2014, Mar.). The JavaScript Object Notation (JSON) data interchange format. RFC 7159. [Online]. Available: http://tools.ietf.org/html/rfc7159

[41] Koverse. (n.d.). Koverse manual version 1.0. [Online]. Available: http://koverse.com/downloads/koverse-manual-1.0.0.pdf

[42] J. Jackson. (2013, Oct. 31). NSA’s Accumulo data store has strict limits on who can see the data. *PC World*. [Online]. Available: http://www.pcworld.com/article/2060060/nsas-accumulo-nosql-store-offers-rolebased-data-access.html

[43] OWASP. (2013, July). Top 10 2013. [Online]. Available: https://www.owasp.org/index.php/Top_10_2013

[44] OWASP. (2014, May). Testing for NoSQL injection. [Online]. Available: https://www.owasp.org/index.php/Testing_for_NoSQL_injection

[45] Cassandra-user mailing list archives. (2011, July 2). Re: CQL injection attacks? [Online]. Available: http://mail-archives.apache.org/mod_mbox/cassandra-user/201107.mbox/%3C1309630640.4e0f60b01d222@itchen.qinetiq.com%3E

[46] Stackoverflow. (2012, Oct. 27). NoSQL injection in Python. [Online]. Available: http://stackoverflow.com/questions/13099301/nosql-injection-in-python

[47] C. Cornutt. (2012, Dec. 19). SQLi in NoSQL - a word of warning. *Websec*. [Online]. Available: http://websec.io/2012/12/19/NoSQL-Injection.html

[48] SCRT Information Security. (2013, Mar. 24). MongoDB - SSJI to RCE. [Online]. Available: http://blog.scrt.ch/2013/03/24/mongodb-0-day-ssji-to-rce/

[49] J. A. Kreibich. (2012, Feb. 29). [Redis] design: Security. *Grokbase*. [Online]. Available: http://grokbase.com/p/gg/redis-db/122x99hdew/redis-re-design-security

[50] B. Sullivan, "Server-side JavaScript injection," White Paper, Adobe, July 2011. [Online]. Available: http://media.blackhat.com/bh-us-11/Sullivan/BH_US_11_Sullivan_Server_Side_WP.pdf

[51] S. Melkote. (2013, Mar. 14). Apache CouchDB traversal arbitrary file access vulnerability. *Securiteam*. [Online]. Available: http://www.securiteam.com/securitynews/5AP37159FI.html

[52] CVE Details. (n.d.). Apache CouchDB security vulnerabilities. [Online]. Available: http://www.cvedetails.com/vulnerability-list/vendor_id-45/product_id-19046/Apache-Couchdb.html

[53] J. A. R. Tunney. (2010, Aug. 13). Tokyo Tyrant protocol vulnerability. *Lobster Technologies*. [Online]. Available: https://web.archive.org/web/20130918232852/http://lobstertech.com/tokyo_tyrant_security_vulnerability.html

[54] Sqrrl. (n.d.). Accumulo. [Online]. Available: http://sqrrl.com/product/accumulo/

[55] Hortonworks. (n.d.). Apache Accumulo. [Online]. Available: http://hortonworks.com/hadoop/accumulo/

[56] *Department of Defense trusted computer system evaluation criteria*, Department of Defense Std. DoD 5200.28-STD, 1985.

[57] S. Castano, M. G. Fugini, G. Martella, and P. Samarati, *Database Security*. Wokingham, England: Addison-Wesley, 1994.

[58] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, May 1976.

[59] R. Sandhu, "Role hierarchies and constraints for lattice-based access controls," in *Computer Security—ESORICS 96*, 1996, pp. 65–79.

[60] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations," MITRE CORP, Bedford, MA, Tech. Rep. MTR-2547-VOL-1, Mar. 1976.

[61] A. Cordova. (2012, Aug.). Jaccson. *GitHub*. [Online]. Available: https://github.com/acordova/jaccson

[62] Apache Thrift. (2014). Thrift network stack. *Apache Software Foundation*. [Online]. Available: https://thrift.apache.org/docs/concepts

[63] R. Graubart, “The integrity-lock approach to secure database management,” in *2012 IEEE Symposium on Security and Privacy*, 1984, p. 62.

[64] T. D. Nguyen, M. A. Gondree, J. Khosalim, and C. E. Irvine, “Towards a cross-domain MapReduce framework,” in *2013 IEEE Military Communications Conference*, 2013, pp. 1436–1441.

[65] I. Roy, S. T. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, “Airavat: Security and privacy for MapReduce,” in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, vol. 10, 2010, p. 20.

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California